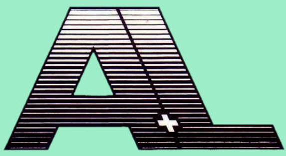
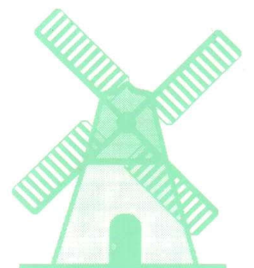


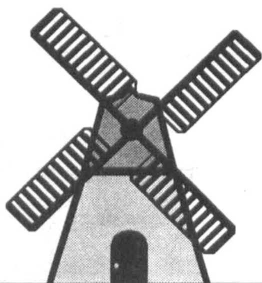
A+L AG

AMIGA OBERON



Amiga Oberon Compiler

Version 3.0



Handbuch

Herstellung und Vertrieb:

A+L AG
Däderiz 61
Ch-2540 Grenchen

Kundenunterstützung

Lesen Sie diese Information, bevor Sie das Siegel des Diskettenumschlags brechen.

Das Siegel des Diskettenumschlags zu brechen, bedeutet gleischzeitig das Akzeptieren der Bedingungen des A+L-Lizenzvertrages.

Die A+L AG und deren Partner helfen Ihnen, Ihr Programm optimal einzusetzen. Wir empfehlen Ihnen, die folgenden Informationen über Registration, Unterstützung und Ihre Rechte und Pflichten als Kunde durchzulesen.

Die A+L AG und deren Partner (dort, wo Sie das Produkt gekauft haben) bieten Ihnen telefonische Unterstützung durch einen Kundendienst. Egal wie kompliziert Ihre Frage oder Ihr Problem ist, wir versuchen Ihnen rasch zu helfen. Bevor Sie jedoch anrufen, vergewissern Sie sich, daß Sie Ihre Registrationskarte bzw. Ihren Lizenzvertrag eingeschickt haben. Andernfalls ist die A+L AG davon entbunden, Ihnen Auskunft zu geben.

Falls Probleme mit Ihrem Programm auftauchen, gehen Sie wie folgt vor:

1. Versuchen Sie, das Problem mit dem Handbuch zu lösen.

2. Rufen Sie A+L oder Ihren Händler zwischen 9 und 17 Uhr an. Sie sollten dann Ihre Programm-Seriennummer zur Hand haben.

Der Autor steht persönlich für Nachfragen nicht zur Verfügung. Probleme, die auf Fehler in der Software zurückzuführen sind, werden jedoch schnellstmöglich an den Autor weitergeleitet.

Beschränkte Garantie

A+L AG garantiert, daß:

1. das Material, Disketten und Dokumentation, nicht beschädigt sind.

2. das Programm ordnungsgemäß auf die Disketten kopiert wurde.

3. die Dokumentation vollständig ist und Informationen enthält, die A+L AG für die Bedienung des Programms für erforderlich hält.

4. das Programm im Prinzip so funktioniert, wie es im Handbuch beschrieben ist.

Falls Sie uns innerhalb von 30 Tagen nach Auslieferung Fehler schriftlich melden oder uns defektes Material zustellen, steht Ihnen im Rahmen dieser beschränkten Garantie das Recht zu, Disketten oder Software ersetzt zu bekommen. Legen Sie der schriftlichen Fehlermeldung das entsprechende Rückporto in Ihrer Währung bei. Das Recht auf Ersatz besteht nur, wenn wir im Besitz Ihres unterschriebenen Lizenzvertrages sind. Senden Sie diesen sowie allfällige Fehlermeldungen oder defekte Disketten an Ihren Händler oder an folgende Adresse:

A+L AG
Däderiz 61
CH-2540 Grenchen, Schweiz

A+L AG haftet nicht für direkte oder indirekte Schäden.

A+L AG lehnt jede Art von Haftung oder stillschweigender Garantie ab, insbesondere, daß sich die Produkte von A+L AG für einen ganz bestimmten Zweck eignen. A+L AG beschränkt ihre Garantie auf das Ersetzen defekter Disketten und Programme.

Entsprechend obenstehender Angaben tragen Sie als Anwender die Verantwortung, mit Ihrem Programm sorgfältig umzugehen, und damit auch die Kosten mutwilliger Beschädigungen und Fehler außerhalb der oben genannten Beschränkungen.

Anwenderlizenz

A+L AG behält sich alle Rechte für ihre Programme vor. Jedes Programm fällt unter diesen Lizenzvertrag. Ein entsprechendes Vergehen wird geahndet.

1. Erlaubter Gebrauch:

Das Programm darf nur auf einem einzigen Computer mit einem einzigen Bildschirm verwendet werden. Sie dürfen eine oder mehrere Kopien dieses Programms herstellen, damit Sie über ausreichend Kopiedisketten verfügen, falls Ihre Arbeitsdisketten zerstört werden. Dazu dürfen Sie eine und nur eine Kopie des Programms auf Ihrer Festplatte machen.

2. Unerlaubter Gebrauch:

Ohne ausdrückliche schriftliche Bewilligung der A+L AG dürfen Sie folgendes nicht tun:

- Das Programm in Verbindung mit mehr als einem Computer gleichzeitig verwenden.
- Das Programm, Kopien davon oder dessen Dokumentation an jemand anderen verkaufen.
- Erstellen von Kopien des Programms, der Dokumentation oder von Disketten, ausgenommen derjenigen Kopien, die in diesem Lizenzvertrag bewilligt sind.

- Die Verwendung des Programms in einem Netz, einem Time-Sharing System, einem interaktiven Fernsehnetzwerk, einem Mehrprozessorensystem oder einem Mehrplatzsystem, falls nicht eine spezielle Lizenz von A+L vorliegt, beziehungsweise jeder einzelne Benutzer über eine eigene Lizenz mit A+L verfügt.

- Veränderungen am Programm vornehmen.

- Das Übertragen des Programms oder das Bewilligen einer Unterlizenz, Vermietung oder das Übertragen weiterer Rechte auf andere.

- Eine wörtliche oder übertragene Übersetzung des Programms herstellen.

- Anpassen des Programms an eine andere Hardware.

- Durchführen telefonischer oder elektronischer Datenübertragung des Programms.

- Vertrieb oder Vermietung des Programms an andere auf dauernder oder auch nur vorübergehender Basis.

- Das Programm oder damit erzeugte Programme oder Daten direkt oder indirekt militärisch nutzen.

- Das Programm oder damit erzeugte Programme oder Daten in lebensnotwendigen oder lebensgefährdenden Systemen einsetzen.

Beendigung der Lizenz:

Die Lizenz gilt als beendet, wenn Sie alle Disketten, Dokumente und Kopien aller Art zerstören. Diese Lizenz fällt ebenfalls dahin, wenn Sie gegen eine der oben genannten Bedingungen verstoßen. Sie sind in einem solchen Fall verpflichtet, alle Ihre Disketten, Dokumente und Kopien aller Art zu vernichten.

Vorwort

Vor Ihnen liegt die neueste Version des Amiga Oberon Compilers. Diese Version wurde stark überarbeitet und in der Leistungsfähigkeit verbessert. Seitdem Amiga Oberon auf dem Markt ist, hat sich die Sprachenlandschaft weiterentwickelt. Auch Oberon blieb davon nicht unberührt, so daß es unterdessen ein erweitertes Oberon mit der Bezeichnung Oberon-2 gibt.

Die Neuerungen in Oberon-2, wie beispielsweise Prozeduren, die wie Methoden anderer objektorientierter Sprachen vererbt werden, machen Oberon erst zu einer leistungsfähigen objektorientierten Sprache. Dies ist auch der Hauptgrund dafür, daß Amiga Oberon um die in Oberon-2 neu eingeführten Funktionen erweitert wurde.

Bisherigen Versionen von Amiga Oberon fehlte die für Oberon eigentlich übliche Speicherverwaltung über einen Garbage-Collector. Da die objektorientierte Programmierung unter dem Fehlen des Garbage-Collectors stark leiden kann, bietet diese Compilerversion die Möglichkeit, den Speicher über einen inkrementellen Garbage-Collector zu verwalten. Wer die gewohnte Speicherverwaltung vorzieht, kann dies mit einer neuen Compileroption wählen.

Um die Programmentwicklung mit Amiga Oberon zu beschleunigen und auch dem Anfänger das Schreiben komplexerer Programme zu erleichtern, wurde die Modulbibliothek um viele Module erweitert. Auch werden von vielen Modulen jetzt die Fähigkeiten von Oberon-2 unterstützt, was die Wiederverwendbarkeit der Module deutlich verbessert.

Außer dem Compiler selbst wurden auch viele der zusätzlichen Programme von Amiga Oberon überarbeitet und um neue Funktionen ergänzt. Besonders betroffen hiervon ist das Programm, mit dem man beim Programmieren den häufigsten und längsten Kontakt hat: der Editor. Er ist unterdessen vollkommen frei konfigurierbar und mit

weit über hundert ARexx-Kommandos frei programmierbar, so daß er an jede Aufgabe ideal angepaßt werden kann.

All diese Neuerungen haben dafür gesorgt, daß das Handbuch vollkommen überarbeitet werden mußte, da es nur noch einen Teil der Funktionen der Programme beschrieben hat. Bei der Gelegenheit wurde das Handbuch auch gleich neu strukturiert und übersichtlicher gestaltet.

Viel Spaß beim Arbeiten mit Amiga Oberon wünschen Ihnen

Fridtjof Siebert und die A+L AG.

Der Amiga Modula und Oberon Klub Stuttgart (AMOK):

Um mit einem Compiler effektiv arbeiten zu können ist es von Vorteil, wenn man auf eine große Modulbibliothek zurückgreifen kann, da man sich dann eine Menge an Entwicklungs- und Programmierarbeit ersparen kann. Zudem können Beispielprogramme vielen bei der Lösung von Problemen helfen.

Der Amiga Modula und Oberon Klub Stuttgart (AMOK) hat sich zum Ziel gesetzt, die Verbreitung der Sprachen Oberon und Modula zu unterstützen. Diese Sprachen haben den großen Vorteil, daß größere Programmprojekte durch Verwendung einer Modulbibliothek sehr viel einfacher realisierbar werden. AMOK ist ein Zusammenschluß von einigen Programmierern, die dies im Sommer 1988 erkannt und so die AMOK PD-Diskettenserie gegründet haben. Sie enthielt zunächst nur Programme und Module der Klubmitglieder. Unterdessen haben jedoch auch viele außenstehende Programmierer ihre Module und Programme freigegeben und uns zur Veröffentlichung überlassen. Dabei legt AMOK großen Wert auf die Qualität und Korrektheit (nicht unbedingt die Komplexität) der Module und veröffentlicht nur einen Teil des eingeschickten Materials.

Unterdessen hat sich eine mehrere Megabyte umfassende Bibliothek von Modula- und Oberon-Modulen angesammelt, die jedem zur Verfügung steht. Für viele Probleme können Lösungen auf den AMOK Disketten gefunden werden.

Um diese Idee auch weiter bestehen zu lassen, sind wir natürlich auf die Einsendung von Modulen und Programmen angewiesen. Wer also ein Modul geschrieben hat, von dem er denkt, daß es auch anderen von Nutzen sein kann, sollte es allen zur Verfügung stellen. Dadurch macht man sich nicht nur einen guten Namen in der Programmiererszene, sondern kann auch Kontakte zu anderen Programmierern knüpfen und so von der Veröffentlichung profitieren. Natürlich ist auch die

Veröffentlichung von sogenannter Shareware möglich, die dem Autor einen kleinen Nebenverdienst einbringen kann.

Die AMOK-Disketten können bei allen guten PD-Vertrieben und bei der A+L AG bezogen werden. Die Klubmitglieder freuen sich immer auf neue Programme und Module. AMOK-Mitglieder sind:

Nicolas Benezan
Postwiesenstr. 2
7000 Stuttgart 60

Kai Bolay
Hoffmannstr. 168
7250 Leonberg

Bernd Kirschner
Gottlob-Grotz-Str. 24
7120 Bietigheim-Bissingen

Fridtjof Siebert
Nobileweg 67
7000 Stuttgart 40

Dabei sollten die Einsendungen möglichst auf alle Mitglieder verteilt werden, um den Arbeits- und Zeitaufwand des einzelnen gering zu halten. Wir sind alles 'nebenberufliche' AMOKler, die ihre Freizeit und damit einen Teil der Zeit, in der sie sonst wohl vor allem programmieren würden, für AMOK opfern. Wir geben keine telefonischen Auskünfte.

Wir können nicht garantieren, daß wir jeden Brief beantworten. Diejenigen Briefe werden meist bevorzugt, die ausreichend Rückporto enthalten. Da wir unsere Disketten nicht in großem Stil kommerziell vertreiben wollen und können, sind wir auf Spenden angewiesen.

Auch in Nürnberg und Würzburg haben sich AMOK-Gruppen gebildet, an die Sie sich wenden können. Ansprechpersonen sind:

Hartmut Goebel
Aufseßplatz 5
8500 Nürnberg 40

Michael Hohmann
Carl-Schilling-Straße 10
8701 Kirchheim / Würzburg

Inhalt

1. Das erste Oberon-Programm

2. Installation

3. Arbeiten mit dem Editor OEd

4. Der Editor OEd

7. Anzeigen von Fehlern mit OErr

8. Das Hilfsprogramm ModToDef

9. Das Make-Utility OMake

10. Der ResidentManager

11. Der Library-Linker LibLink

12. Unterschiede zwischen Modula-2 und Oberon

13. Unterschiede zwischen Oberon und Oberon-2

14. Besonderheiten des Compilers

15. Amigaspezifische Erweiterungen

19. Modulbibliothek: Datenstrukturen

20. Modulbibliothek: Zeichenkettenbearbeitung

21. Modulbibliothek: Mathematikmodule

22. Modulbibliothek: Ein- und Ausgabe

23. Modulbibliothek: Standardmodule

24. Modulbibliothek: Multitasking

25. Modulbibliothek: Amigaspezifische Module

26. Modulbibliothek: Oberon-Unterstützung

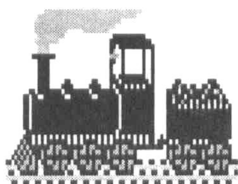
27. Benutzung der AmigaOS-Interface-Module

Anhang A: Fehlermeldungen

Anhang B: Syntax von Amiga Oberon

Anhang C: Tabellen

1. Das erste Oberon-Programm



Dieses Kapitel ist für all diejenigen, die es nicht abwarten können, diese Anleitung vollständig zu lesen, bevor sie ihr erstes Programm schreiben. Wer gleich mit diesem Oberon-Entwicklungssystem arbeiten möchte und einen groben Überblick über die wichtigsten Programme bekommen möchte, sollte hier weiterlesen. Geduldigere können den Rest dieses Kapitels überspringen und in Ruhe mit Kapitel 2 und der Installation des Systems beginnen.

Um ohne viel Installationsaufwand den Compiler benutzen zu können, muß 'QuickInstall' der ersten Diskette gestartet werden. Dies geschieht am einfachsten durch einen Doppelklick auf das gleichnamige Piktogramm. 'QuickInstall' kopiert ein paar Dateien in das logische Verzeichnis 'LIBS:' und in die RAM-Disk. Dazu müssen Sie nacheinander verschiedene Disketten des Oberon-Systems einlegen.

Dann starten Sie den Editor OEd von der ersten Diskette durch einen Doppelklick auf sein Piktogramm. Wer schon einmal mit einem Texteditor gearbeitet hat, wird in der Bedienung kaum Probleme haben. Nun kann ein der Text eines Oberon-Programms eingegeben werden. Unser Vorschlag ist folgendes 'Hallo Welt'-Beispielprogramm:

```
MODULE Hallo;

IMPORT io;

VAR
  a,b: INTEGER;

  PROCEDURE Space(n: INTEGER);
  BEGIN
    WHILE n>0 DO
```

```
      io.Write(" ");
      DEC(n);
    END;
  END Space;

BEGIN
  a := 0;
  b := 32;
  REPEAT
    Space(32-a); io.WriteString("Hallo");
    Space(2*a ); io.WriteString(" Welt!");
    io.WriteLn;
    INC(a,b DIV 5);
    DEC(b,a DIV 5);
  UNTIL a<0;
END Hallo.
```

Haben Sie das Programm vollständig eingegeben, sollten Sie es speichern. Dazu wählen Sie im Projekt-Menü des Editors den Menüpunkt 'Save As' aus und geben anstelle des Namens 'unnamed' den Namen 'RAM:Hallo.mod' ein. Nach einem Klick auf das 'Ok'-Symbol wird der Text in die RAM-Disk gespeichert. Der Editor durch nun durch einen Klick auf das Schließsymbol seines Fensters verlassen. Mit der Workbench sollte man sich den Inhalt der RAM-Disk ansehen. Dort befindet sich ein Piktogramm mit dem Namen 'Hallo.mod'.

Das Windmühlen-Piktogramm mit dem Namen 'Oberon' auf der ersten Oberon-Diskette ist der Compiler. Wird dieser einmal angeklickt und der Text in der RAM-Disk bei gedrückter Shift-Taste doppelgeklickt, beginnt der Compiler seine Arbeit und übersetzt den Text.

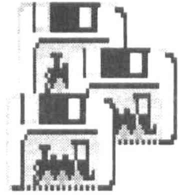
Danach enthält die RAM-Disk zusätzlich die Dateien 'Hallo.sym' und 'Hallo.obj'. Letztere wird einfach durch einen Doppelklick auf ihr Piktogramm in ein ausführbares Programm umgewandelt. Es wird das

Programm 'OLink' ausgeführt, das das ausführbare Programm 'Hallo' erzeugt.

Das erzeugte Programm kann leicht an dem Piktogramm mit der Dampflokomotive erkannt werden. Durch einen Doppelklick wird es ausgeführt.

Sind Sie überrascht über die Ausgabe? Man kann auch mit recht einfachen Mitteln interessante Effekte erzielen!

2. Installation



Hardwareanforderungen

Um diesen Compiler sinnvoll benutzen zu können, sollte man einen Amiga mit mindestens zwei Mega-Byte Arbeitsspeicher besitzen. Eine Festplatte wird dringend empfohlen, mit sehr viel Geduld und Ausdauer beim Diskettenwechseln kann man jedoch auch mit zwei Diskettenlaufwerken auskommen. Amiga Oberon benötigt die Betriebssystemversion AmigaOS 1.2 und mindestens die Workbench 1.3. Es wird jedoch empfohlen, unter AmigaOS 2.04 oder neueren Versionen zu arbeiten.

Vorbereitung

Vor der Installation sollten zunächst Sicherheitskopien der Originaldisketten hergestellt werden. Nachdem diese an einen sicheren, vor Umwelteinflüssen (Kaffeemaschinen, Lautsprechern, kleinen Kindern, etc.) geschützten Ort verstaut sind, kann man sich eine Tasse Kaffee holen und mit der Installation beginnen.

Da Sie nach der Installation womöglich ein paar Stunden mit dem Oberon-System arbeiten werden und alle Möglichkeiten gleich am ersten Tag ausprobieren wollen, sollten Sie sich genügend Lebensmittel rund um den Computer bereitstellen und sich für eine Weile von ihrer Familie verabschieden.

Installation auf Harddisk

Im folgenden wird davon ausgegangen, daß die Arbeitspartition der Harddisk den Namen 'Work' hat. Wurde sie anders benannt, ist in den beschriebenen Anweisungen der Text 'Work' durch den Namen der entsprechenden Partition zu ersetzen.

Zunächst muß ein Verzeichnis auf der Arbeitspartition erzeugt werden, in das die Programme und Dateien von Amiga Oberon kopiert werden sollen. Dieses Verzeichnis bekommt sinnvollerweise den Namen 'OBERON'. Um es zu erzeugen, muß folgende Anweisung in ein Shell-Fenster eingegeben werden:

```
MAKEDIR Work:OBERON
```

In dieses Verzeichnis sollten nun alle Dateien der Amiga Oberon Disketten kopiert werden. Dabei muß darauf geachtet werden, daß das Datum der Dateien nicht verändert wird, da sonst das Make-Utility nicht korrekt arbeiten kann. Werden die Disketten nacheinander in das Diskettenlaufwerk 'DF0:' eingelegt, so können die Dateien mit der Anweisung

```
COPY DF0: TO Work:OBERON ALL CLONE
```

kopiert werden. Bei Platzmangel auf der Harddisk kann auf die Quelltexte in den Verzeichnissen 'Interfaces' und 'Module' verzichtet werden. Zudem können die Objektdateien mit den Endungen '.obja' und '.objsa' aus dem Unterverzeichnis 'obj' gelöscht werden, wenn der Garbage-Collector bei der Compilation von Programmen nicht ausgeschaltet wird. Die Objektdateien mit den Endungen '.objs' und '.objsa' werden nicht benötigt, wenn das kleine Datenmodell nicht verwendet wird. Genaueres hierzu ist in den Kapiteln 14 und 16 zu finden.

Die Programme von Amiga Oberon greifen auf das Verzeichnis 'OBERON:' zu, wenn sie auf bestimmte Dateien oder auf andere Programme zugreifen. Daher muß die Datei 'S:User-Startup' oder auf Rechnern, die noch mit AmigaOS 1.3 oder noch älteren Versionen des AmigaOS arbeiten, die Datei 'S:Startup-Sequence' um die folgende Anweisung ergänzt werden:

```
ASSIGN OBERON: Work:OBERON
```

Auf Amigas mit einem großen Arbeitsspeicher sollten oft benötigte Programme von Amiga Oberon speicherresident gehalten werden, damit sie schneller gestartet werden können und ein flüssigeres Arbeiten möglich wird. Um die Programme Oberon, OLink und OEd automatisch resident zu halten, sollte die Datei 'S:User-Startup' bzw. 'S:Startup-Sequence' noch um die folgenden Anweisungen erweitert werden:

```
RESIDENT OBERON:Oberon
RESIDENT OBERON:OLink
RESIDENT OBERON:OEd
```

Soll mit Amiga Oberon nicht nur von der Workbench-Oberfläche sondern auch von der Shell aus gearbeitet werden, ist es sinnvoll, die Datei 'S:Shell-Startup' um die Anweisung

```
PATH OBERON: ADD
```

zu ergänzen. Dann können die Programme von Amiga Oberon ohne Angabe ihres Pfades direkt gestartet werden.

Die Amiga Oberon Programme suchen die Objekt- und Symboldateien in den Verzeichnissen, deren Pfade in der Textdatei 'OBERON:Path' aufgelistet sind. Das Make-Utility OMake (siehe Kapitel 9) sucht dort auch die Quelltexte. Dabei enthält 'OBERON:Path' in jeder Zeile einen Pfad.

Gewöhnlich sind die Objekt- und Symboldateien alle im Verzeichnis 'OBERON:' enthalten, so daß es ausreichend ist, wenn 'OBERON:Path' aus der folgenden Zeile besteht:

```
OBERON:
```

Werden einige Objekt- oder Symboldateien in anderen Verzeichnissen gespeichert, so sollte auch der Name dieses Verzeichnisses in 'OBERON:Path' angegeben werden.

Damit die Programme von Amiga Oberon arbeiten können, benötigen sie zwei Funktionsbibliotheken, die in das Verzeichnis 'LIBS:' der Systempartition kopiert werden müssen: die 'oberonsupport.library' und die 'garbagecollector.library'. Die zweite der beiden Dateien wird auch von allen mit Amiga Oberon übersetzten Programmen benötigt, die den Garbage-Collector verwenden. Um diese Dateien zu kopieren sind die folgenden beiden Anweisungen in ein Shell-Fenster einzugeben:

```
COPY Work:OBERON/libs/oberonsupport.library LIBS:
COPY Work:OBERON/libs/garbagecollector.library LIBS:
```

Auf Amigas, die noch unter AmigaOS 1.3 oder älteren Versionen des AmigaOS arbeiten, muß zusätzlich die Bibliothek 'arp.library' installiert werden. Der Editor OEd benötigt diese Bibliothek. Sie wird mit der folgenden Anweisung kopiert:

```
COPY Work:OBERON/libs/arp.library LIBS:
```

Nach einem Neustart des Computers kann nun begonnen werden, mit Amiga Oberon zu arbeiten. Statt einem Neustart reicht es jedoch auch, die beiden Anweisungen

```
ASSIGN OBERON: Work:OBERON
PATH OBERON: ADD
```

in ein Shell-Fenster einzugeben.

Installation ohne Harddisk

Steht keine Harddisk zur Verfügung, kann mit den Disketten des Amiga Oberon Compilers zunächst ohne Änderungen gearbeitet werden. Um unnötig viele Diskettenwechsel zu vermeiden sollten die gewöhnlich benötigten Programme und Dateien auf wenigen Disketten ge-

sammelt werden. Leider ist es schwer möglich im voraus zu sagen, welche Dateien von einem bestimmten Amiga Oberon Programmierer benötigt werden. Welche Objekt- und Symboldateien verwendet werden hängt vor allem stark davon ab, welche Art von Programmen mit Amiga Oberon geschrieben werden sollen. Auch ist hierbei wichtig, ob der Garbage-Collector oder das kleine Datenmodell benutzt werden oder ob dies nicht der Fall ist.

Nachdem Sie eine Weile mit Amiga Oberon gearbeitet haben, sollten Sie sich daher Disketten für das Programmieren mit Oberon zusammenstellen, die die für Ihre Anwendungen wichtigen Dateien und Programme enthalten.

Auf der Workbench-Diskette muß die Datei 'S:User-Startup', bzw. die Datei 'S:Startup-Sequence' auf Rechnern, die unter AmigaOS 1.3 oder älteren Betriebssystemversionen arbeiten, um die Anweisung

```
ASSIGN OBERON: AmigaOberon3.0_Disk1:
```

ergänzt werden (Bei neueren Versionen von Amiga Oberon ist statt '3.0' natürlich die aktuelle Versionsnummer einzugeben). Diese Anweisung teilt den Programmen von Amiga Oberon mit, daß sie wichtige Dateien und Programme auf der ersten Diskette von Amiga Oberon finden.

Soll mit Amiga Oberon nicht nur von der Workbench-Oberfläche sondern auch von der Shell aus gearbeitet werden, ist es sinnvoll, die Datei 'S:Shell-Startup' auf der Workbench-Diskette um die Anweisung

```
PATH OBERON: ADD
```

zu ergänzen. Dann können die Programme von Amiga Oberon ohne Angabe ihres Pfades direkt gestartet werden.

Außer den Änderungen an diesen Dateien müssen für das Arbeiten mit Amiga Oberon noch zwei Funktionsbibliotheken in das Verzeichnis

'LIBS:' der Workbench-Diskette kopiert werden. Es handelt sich hierbei um die Dateien 'oberonsupport.library' und 'garbagecollector.library'. Die zweite der beiden Dateien wird auch von allen mit Amiga Oberon übersetzten Programmen benötigt, die den Garbage-Collector verwenden. Um diese Dateien zu kopieren sind die folgenden beiden Anweisungen in ein Shell-Fenster einzugeben:

```
COPY AmigaOberon3.0_Disk1:libs/oberonsupport.libra
ry TO LIBS:
COPY AmigaOberon3.0_Disk1:libs/garbagecollector.li
brary TO LIBS:
```

Auf Amigas, die noch unter AmigaOS 1.3 oder älteren Versionen des AmigaOS arbeiten, muß zusätzlich die Bibliothek 'arp.library' installiert werden. Der Editor OEd benötigt diese Bibliothek. Sie wird mit der folgenden Anweisung auf die Workbench-Diskette kopiert:

```
COPY AmigaOberon3.0_Disk1:libs/arp.library LIBS:
```

In der Textdatei 'Path' der ersten Diskette von Amiga Oberon werden die Pfade angegeben, in denen die Programme von Amiga Oberon die Objekt- und Symboldateien suchen. Zudem benutzt das Make-Utility diese Pfade, um nach Quelltextdateien zu suchen.

Da diese Dateien über mehrere Disketten von Amiga Oberon verteilt sind, sollten hier die Namen aller Amiga Oberon Disketten jeweils gefolgt von einem Doppelpunkt eingetragen werden. Dabei muß jeder Pfad in einer eigenen Zeile stehen. Die Datei sollte also in etwa folgendermaßen aussehen:

```
AmigaOberon3.0_Disk1:
AmigaOberon3.0_Disk2:
usw.
```

Nach einem Neustart oder der Eingabe der beiden Anweisungen

```
ASSIGN OBERON: AmigaOberon3.0_Disk1:
PATH OBERON: ADD
```

in ein Shell Fenster kann mit Amiga Oberon programmiert werden.

3. Arbeiten mit dem Editor OEd



Dieses Kapitel enthält eine kurze Anleitung zum Editor OEd. Sie soll Ihnen helfen, mit dem Editor vertraut zu werden, ohne gleich von seiner Fülle an Fähigkeiten und Kommandos erschlagen zu werden. Scheuen Sie nicht davor zurück, sich gleich intensiv mit dem Editor auseinanderzusetzen, können Sie dieses Kapitel überspringen und mit Kapitel 4 fortfahren. Möchten Sie dagegen nur einen groben Überblick über die Fähigkeiten des Editors und möchten Sie danach gleich mehr über den Compiler und die anderen Programme dieses Oberon-Entwicklungssystems erfahren, können Sie nach der Lektüre dieses Kapitels gleich mit Kapitel 5 und den folgenden Kapiteln fortfahren.

Installation

Damit OEd arbeiten kann, müssen sich die Dateien 'OEd_Keys.txt' und 'OEd_Menu.txt' in dem Verzeichnis 'OBERON:' befinden. Auf Amigas, die noch unter AmigaOS 1.3 oder noch älteren Versionen des AmigaOS arbeiten, benötigt OEd zudem die arp.library im Verzeichnis 'LIBS:'. Viele Kommandos benötigen ARexx, so daß ARexx korrekt installiert sein sollte. Genaueres dazu im nächsten Kapitel.

Starten von OEd

Aufruf von der Workbench

Gestartet wird der Editor durch einen Doppelklick auf sein Piktogramm. Soll gleich ein Text geladen werden, so kann dieser zuvor angeklickt und danach OEd bei gedrückter Umschalttaste (Shift) doppelgeklickt werden.

Aufruf von der Shell

Wird von der Shell aus gearbeitet, so wird OEd einfach durch die Ein-

gabe von *OEd* in die Shell gestartet. Soll die Shell weiter benutzt werden, solange OEd läuft, ist der Aufruf mit *run OEd* nötig. Von der Shell können die Namen der zu bearbeitenden Texte als Argumente übergeben werden, so daß *run OEd Test.mod Demo.mod* zwei OEd-Fenster mit den Texten 'Test.mod' und 'Demo.mod' öffnet.

Gewöhnliches Arbeiten mit OEd

Der Editor verhält sich ähnlich wie viele andere Texteditoren für den Amiga auch. Eine Schreibmarke (Cursor) markiert die aktuelle Position im Text. Gewöhnliche Tasten fügen das ihnen zugeordnete Zeichen an der aktuellen Position ein. Mit der Return-Taste wird eine Zeile unter der aktuellen Zeile eingefügt und die Schreibmarke wird in diese Zeile gesetzt.

Die Rück- und die Löschtaste löschen das Zeichen links neben bzw. unter der Schreibmarke. Die Umschalttaste zusammen mit diesen Tasten gedrückt löschen die aktuelle Zeile. Alt zusammen mit diesen Tasten löschen alle Zeichen links bzw. rechts und unter der Schreibmarke bis zum nächsten Wortanfang. Steuerung (Ctrl) und diese Tasten löschen alle Zeichen der aktuellen Zeile links bzw. rechts und unter der aktuellen Position.

Mit den Pfeiltasten kann die Schreibmarke zeichenweise bewegt werden. Zusammen mit der Umschalttaste wird sie an den Anfang bzw. an das Ende des Textes (Tasten hoch und runter) oder der aktuellen Zeile (Tasten links und rechts) gesetzt. Alt und die Pfeiltasten bewegen die Schreibmarke seiten- bzw. wortweise in die angegebene Richtung. Steuerung und diese Tasten bewegen den Text um eine Position wobei die Schreibmarke ihre Position auf dem Bildschirm beibehält.

Mit den Symbolen am rechten Rand des OEd-Fensters kann man sich auch leicht durch den Text bewegen.

Das Project-Menü

Project	
Load...	L
New Window...	W
Save	S
Save as...	V
Save Block...	O
Insert File...	I
Print	Ctrl+p
Print Block	Ctrl+b
Print As...	Ctrl+i
Print Block As...	Ctrl+o
Hide	Ctrl+Esc
Reveal...	Ctrl+v
Screen Mode...	Ctrl+d
About...	J
Quit	Q

Load... öffnet ein Dateiauswahlfenster, mit dem eine neue Datei geladen werden kann. *New Window...* öffnet ein weiteres, leeres Textfenster. Mit *Save* und *Save as...* wird der Text unter dem alten bzw. einem neuen Namen gespeichert. Der markierte Block (siehe unten) wird mit *Save Block...* in eine Datei gespeichert, *Insert File...* fügt eine Datei über der aktuellen Zeile ein.

Mit den *Print*-Menüpunkten kann der Text bzw. der markierte Block ausgedruckt werden. Bei *Print As...* und *Print Block As...* können dabei noch zusätzlich die Druckeinstellungen verändert werden.

Sind mehrere Textfenster geöffnet, so können einzelne zeitweise mit *Hide* geschlossen und später mit *Reveal...* wieder angezeigt werden. Dazu öffnet *Reveal...* ein Dialogfenster, in dem der gewünschte Text ausgewählt werden kann.

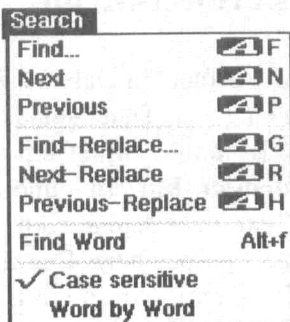
Unter AmigaOS 2.0 kann mit *Screen Mode...* der Bildschirmmodus gewählt werden, wenn OEd einen eigenen Bildschirm öffnet (siehe Kapitel 5).

About... zeigt die Version von OEd und Informationen über das Copyright an. *Quit* schließt das aktive OEd-Fenster.

Suchen und Ersetzen

Mit den Punkten des *Search*-Menüs kann nach Text gesucht und dieser auf verschiedene Arten durch einen anderen Text ersetzt werden. *Find...* öffnet ein Dialogfenster, mit den ein Suchtext eingegeben wer-

den kann. Zudem kann hier gewählt werden, ob Groß- und Kleinschreibung unterschieden werden soll (*Case-Sensitiv*), ob vorwärts oder rückwärts und ob wortweise oder zeichenweise gesucht werden soll. Wird das Dialogfenster mit dem Symbol 'ok' verlassen, so wird die Schreibmarke auf das nächste Vorkommen des Suchtextes gesetzt.



Next und *Previous* suchen nach dem nächsten bzw. vorherigen Vorkommen des Suchtextes und setzen die Schreibmarke an die Position des gefundenen Textes.

Suchen und Ersetzen ist mit *Find-Replace...* möglich. Es wird ein Dialogfenster geöffnet, das außer den Symbolen des *Find...*-Dialogfensters noch die Eingabe eines Ersatztextes erlaubt und die Wahl zwischen *alle* und *einzeln* ersetzen ermöglicht. Nach dem Beenden des Dialogfensters mit dem Symbol 'ok' wird nach dem nächsten Vorkommen des Suchtextes gesucht. War *einzeln* angewählt, so wird dieses durch den Ersetztext ersetzt und danach die Operation abgebrochen. War dagegen *alle* gewählt, so wird in einem zweiten Dialogfenster gefragt, ob der gefunden Text ersetzt werden soll (*ja!*) oder nicht (*nein!*), ob alle folgenden Vorkommen ohne Rückfrage ersetzt werden sollen (*alle*) oder ob die Operation abgebrochen werden soll (*beenden*). Damit die Symbole nicht umständlich mit der Maus angefahren werden müssen, genügt hier auch jeweils die Eingabe des Anfangsbuchstabe, also 'j', 'n', 'a' oder 'b' von der Tastatur.

Next-Replace und *Previous-Replace* ersetzen das nächste Vorkommen des Suchtextes hinter bzw. vor der Schreibmarke.

Mit *Find Word* kann nach dem nächsten Vorkommen des Wortes unter der Schreibmarke gesucht werden.

Die letzten beiden Menüpunkte *Case sensitive* und *Word by Word* ent-

sprechen den Symbolen *Case-Sensitiv* und *wortweise* des *Find*-Dialogfensters.

Blöcke bearbeiten

Block	
Begin	B
End	E
Mark All	Ctrl+k
Copy Block	Y
Move	M
Delete	D
Cut	Alt+x
Copy	Alt+c
Paste	Alt+v
TAB right	Ctrl+r
TAB left	Ctrl+l
Unmark	U

Ein Block ist ein Teil des Textes, der durch eine besondere Markierung vom Rest des Textes hervorgehoben wird. Ein Block kann mit der Maus durch Anwählen des Anfangs, gedrückt halten der linken Maustaste, Anfahren des gewünschten Endes und Loslassen der Maustaste markiert werden. Ein andere Möglichkeit besteht darin, den Blockanfang mit der Schreibmarke anzufahren *Begin* aus dem Block-Menü anzuwählen, die Schreibmarke auf das Ende zu setzen und *End* anzuwählen. Soll der gesamte Text als Block markiert werden, reicht es, *Mark All* anzuwählen.

OEd unterscheidet zwischen mehrzeiligen Blöcken und horizontalen Blöcken, die sich nur über einen Teil einer Zeile erstrecken.

Copy Block kopiert den markierten Block an die aktuelle Position. Ein mehrzeiliger Block wird über der aktuellen Zeile eingefügt. *Move* verschiebt den markierten Block von seiner ursprünglichen Position zur aktuellen Position. *Delete* löscht den markierten Block.

Cut und *Copy* kopieren den Block in den Zwischenspeicher des Clipboards. *Cut* löscht ihn zudem aus dem Text. *Paste* fügt den Block aus dem Zwischenspeicher an der aktuellen Position wieder ein. Auf diese Weise kann ein Block auch zwischen mehreren Textfenstern kopiert werden.

TAB right und *TAB left* verschieben den markierten mehrzeiligen Block über eine Tabulatorweite nach rechts bzw. links.

Unmark löscht die Blockmarkierung.

Das Special-Menü

Special	
Enter Rexx Cmd...	Ctrl+e
Execute Rexx Cmd	Ctrl+x
Stop Rexx (HI)	Ctrl+c
Goto Line...	Ctrl+g
Goto Last Change	Ctrl+a
Matching Bracket	Ctrl+h
Change Case	Alt+a
Undo	Ctrl+<
Redo	Ctrl+>
Undo All	
Redo All	
Clear All Undos	Ctrl+z
Undelete Line	Alt+l

Mit *Enter Rexx Cmd...* kann ein kleines ARexx-Programm eingegeben und ausgeführt werden, siehe Kapitel 5. *Execute Rexx Cmd* führt das bei *Enter Rexx Cmd...* eingegebene ARexx-Programm nochmals aus. *Stop Rexx (HI)* startet das Programm 'HI', daß alle ARexx-Programm abbricht. Dies kann dazu verwendet werden, ein fehlerhaftes ARexx-Programm, das bei *Enter Rexx Cmd...* eingegeben wurde, anzuhalten.

Goto Line... öffnet ein Dialogfenster, in das eine Zeilennummer eingegeben werden kann. Wird das Fenster mit dem Symbol 'ok' verlassen, so wird die Schreibmarke in die Zeile mit dieser Nummer gesetzt. *Goto Last Change* setzt die Schreibmarke an die Position, an der sie war, als die letzte Änderung am Text vorgenommen wurde.

Befindet sich die Schreibmarke auf einer runden, eckigen oder geschweiften Klammer, so wird sie von *Matching Bracket* auf die entsprechende (öffnende bzw. schließende) Klammer gesetzt.

Change Case wandelt das Zeichen unter der Schreibmarke von Groß- in Kleinschreibung und umgekehrt.

Undo macht die letzte Textänderung rückgängig, *Redo* macht das letzte *Undo* rückgängig. *Undo All* macht alle gespeicherten Textänderungen seit dem letzten Speichern rückgängig (gewöhnlich werden 50 Änderungen gespeichert, im fünften Kapitel wird beschrieben, wie man diese Zahl erhöht). *Redo All* macht alle zuvor ausgeführten *Undo* und *Undo All* rückgängig. *Clear All Undos* kann bei Speichermangel verwendet werden. Es gibt den für das Merken der Textänderungen nötigen Speicher frei. Danach ist *Undo* und *Redo* der bisherigen Textänderungen nicht mehr möglich.

Undelete Line fügt die zuletzt mit der Umschalttaste und der Rück- bzw. der Löschtaste gelöschte Zeile über der aktuellen Zeile ein.

Makros

Macros	
Start Learning	Shift+F 10
Stop Learning	Ctrl+f
Play Macro	F 10

Ein Makro ist eine kurze Folge von Editorbefehlen, die beliebig ausgeführt werden können. Mit *Start Learning* beginnt OEd alle Eingaben als Makro aufzuzeichnen. *Stop Learning* schließt diese Aufzeichnung ab. Danach kann das erzeugt Makro mit *Play Macro* beliebig oft wiederholt abgespielt werden.

Beispiel: Nehmen wir an, Sie müßten zehnmal *'hallo'* schreiben. Dies geschieht am einfachsten mit einem kleinen Macro. Die wählen also *Start Learning*, schreiben *'hallo'* und wählen *Stop Learning*. Nun brauchen Sie lediglich noch neunmal *Play Macro* auszuführen.

Oberon-Unterstützung

Oberon	
Parse	A
Compile...	C
Link...	K
Make...	Ctrl+m
Execute...	X
Compiler Options...	Ctrl+t
Next Error	T
First Error	Z
Reload Errorfile	Ctrl+q

Der Editor soll vor allem die Programmierung in Oberon erleichtern. Daher bietet er die folgenden Menüpunkte an:

Parse untersucht den Text, ob er der Syntax von Amiga Oberon entspricht (siehe Anhang B). Ist dies nicht der Fall, so wird eine entsprechende Fehlermeldung in der Titelzeile des OEd-Fensters angezeigt. *Compile...*, *Link...* und *Make...* starten den Compiler, den Linker bzw. das Make-Utility mit dem Namen des bearbeiteten Textes als Parameter. *Execute...* startet das vom Linker erzeugt Programm. Mit *Compiler Options...* können Compileroptionen für bedingte Compilation gesetzt werden (siehe Kapitel 14). Traten bei der Compilation Fehler auf, so zeigt *Next Error* die nächste Fehlermeldung im Text hinter der Schreibmarke an. *First Error* zeigt die erste Fehlermeldung im Text. *Reload Errorfile* zwingt OEd, die

Fehlerdatei neu einzulesen. Dies ist z.B. dann nötig, wenn der Text von der Shell aus neu compiliert wurde und dabei eine neue Fehlerdatei erzeugt hat.

Options

✓	StackChk	1
✓	OvflChk	2
✓	RangeChk	3
✓	CaseChk	4
✓	ReturnChk	5
✓	NilChk	6
✓	TypeChk	7
	OddChk	
✓	AutRegPars	
✓	Clear Vars	
✓	New Symbolfile	
	SmallCode	8
	SmallData	9
✓	Garbage-Collector	+
✓	Language Extensions	#
	Debug	
	68010	
	68020	
	68030	
	68881/68882	

Die Compileroptionen

Mit den Punkten des *Options*-Menüs können die in Kapitel 14 beschriebenen Compileroptionen gesetzt bzw. gelöscht werden. Sie werden beim Start des Compilers und des Make-Utilities direkt den Programmen Oberon bzw. OMake übergeben.

Die Optionen *SmallCode*, *SmallData* und *Garbage-Collector* werden beim Linken auch an OLink übergeben.

Einstellungen

Settings

✓	Insert	.
✓	Layout Mode	-
	Auto Uppercase	,
✓	Create Icons?	0
	Set Script Flag?	
	No Umlauts	Ctrl+ß
	Umlauts	Shift+Ctrl+ß

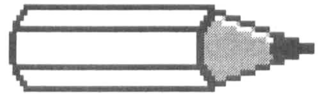
Mit dem *Settings*-Menü können verschiedene Einstellungen gewählt werden: *Insert* wählt zwischen Einfüge- und Überschreibmodus beim Schreiben von Text. *Layout Mode* wählt den Layout-Modus. Ist dieser Modus nicht aktiv, so kann man die Schreibmarke in der ersten Spalte nach links und in der letzten Spalte einer Zeile nach rechts bewegen. Sie wird dann jeweils an das Ende der darüberliegenden bzw. an den Anfang der nächsten Zeile gesetzt.

Soll OEd Piktogramme für die Texte erzeugen, so muß *Create Icons?*

angewählt sein. Wird mit OEd eine Amiga-Skript-Datei geschrieben, so kann *Set Skript Flag?* angewählt werden, damit das Script-Flag beim Speichern der Datei gesetzt wird, die Datei also ausführbar wird.

Wird OEd zum Schreiben von elektronischen Briefen verwendet, so kann mit *No Umlauts* die Tastaturbelegung so geändert werden, daß die Umlaute und das Zeichen 'ß' als 'ae', 'oe' etc. und 'ss' ausgegeben werden. *Umlauts* belegt die Tasten wieder mit ihren ursprünglichen Zeichen.

4. Der Editor OEd



Dieser Editor wurde speziell für diesen und mit diesem Compiler entwickelt und unterstützt besonders die Entwicklung von Oberonprogrammen. Er kann leicht mit der Maus bedient werden, erlaubt das gleichzeitige Bearbeiten mehrerer Texte, enthält einen Oberon-Syntaxchecker, erlaubt das Starten von Compiler, Linker und compiliertem Programm und kann die Fehlermeldungen des Compilers anzeigen. Durch einen leistungsfähigen ARexx-Port und frei belegbare Tasten und Menüs kann OEd jedoch auch an viele andere Anwendungen angepaßt werden.

Starten von OEd

Gestartet wird OEd einfach mit einem Doppelklick auf sein Piktogramm oder durch Eingabe von 'OEd' in einem Shell-Fenster. Werden Argumente übergeben oder wurden von der Workbench zuvor Texte angeklickt und OEd mit gedrückter Umschalttaste (Shift) doppelgeklickt, dann werden die angewählten Texte automatisch in jeweils ein OEd-Fenster geladen.

Aufruf von der Workbench

Vor dem Start von der Workbench können die Voreinstellungen als Merkmale des OEd-Piktogramms (Tool Types) mit der Funktion 'Information' der Workbench gesetzt werden. OEd kennt dabei folgende Merkmale:

Merkmal:	Bedeutung:
ICONS	Piktogramme erzeugen
TABULATOR	Tabulatorweite = n
LEFTEDGE	Horizontale Position der Textfenster
TOPEDGE	Vertikale Position der Textfenster
WIDTH	Breite der Textfenster

```
HEIGHT         Höhe der Textfenster
DEPTH          Tiefe des OEd-Bildschirms
SCREEN         OEd soll eigenen Bildschirm öffnen
INTERLACE      Bildschirm im Interlace-Modus öffnen
MAXUNDO        Größe des 'undo'-Puffers (s.u.)
LAYOUT         Setzt den Schalter 'layout' (s.u.)
AUTOUPPERCASE  Setzt den Schalter 'autouc' (s.u.)
```

Die folgenden Piktogrammerkmale entsprechen denen des Compilers und werden in Kapitel 5 genauer beschrieben:

```
STACKCHK          OVFLCHK     RANGECHK
CASECHK           RETURNCHK   NILCHK
ODDCHK            TYPECHK     AUTOREGPARS
CLEARVARS         SMALLCODE   SMALLDATA
MC68010           MC68020     MC68030
MC68881           DEBUG       NEWSYMFILE
GARBAGECOLLECTOR  EXTENSIONS
```

Aufruf von der Shell

Wird von der Shell aus gearbeitet und sollen mehrere Texte editiert werden, dann ist es meist sinnvoll, zunächst das aktuelle Verzeichnis auf das zu setzen, welches die Texte enthält.

OEd hat folgende Aufrufsyntax:

```
OEd {-{i|t#|x#|y#|w#|h#|d#|s|l|u#|o|a}}
    [c-{svbcrnotzmdl238igyae}] {<Text>}
```

Gefolgt von einem Minuszeichen '-' können Optionen festgelegt werden. Das Nummernzeichen '#' steht jeweils für eine Dezimalzahl.

Die einzelnen Optionen haben die in folgender Tabelle beschriebene Bedeutung:

Option:	Bedeutung
-i	Piktogramme erzeugen
-t#	Tabulatorbreite auf # setzen.
-x#,-y#	x- und y-Position der OEd-Fenster
-w#,-h#	Breite und Höhe der OEd-Fenster
-d#	Tiefe des OEd-Bildschirms
-s	OEd soll eigenen Bildschrim öffnen
-l	Bildschirm im Interlace-Modus öffnen
-u#	Maximale Anzahl der Textänderungen, die sich OEd merken soll und die mit dem Kommando 'undo' (s. u.) rückgängig gemacht werden können. Voreingestellt ist 50.
-o	deaktiviert den Schalter 'layout' (s.u.)
-a	aktiviert den Schalter 'autouc' (s.u.)

Hinter 'c-' können die Optionen angegeben werden, die beim Aufruf des Compilers und des Linkers verwendet werden sollen. Ihre Bedeutung und Voreinstellung ist identisch mit der beim Start des Compilers von der Shell aus (siehe Kapitel 4).

Beispielaufruf:

```
OEd -syl1d1u100 Hello.mod
```

Hier öffnet OEd das Textfenster mit dem Quelltext "Hello.mod" auf einem zweifarbigen Bildschirm direkt unter der Bildschirm-Titelzeile. Beim Editieren merkt sich OEd die letzten 100 Änderungen, so daß diese mit dem Kommando *undo* rückgängig gemacht werden können.

Das folgende Bild zeigt das durch diesen Aufruf geöffnete OEd-Fenster:

```
col: 19 line: 5 of 15 chgs: 1 file: ram:Hello.mod
MODULE Hello;

IMPORT io;

VAR
  i: INTEGER;

BEGIN
  i := 0;
  REPEAT
    io.WriteString("Hello World!");
    INC(i);
  UNTIL i=1000;
  io.WriteLn;
END Hello.
```

Arbeiten mit OEd

Die Titelzeile der OEd-Fenster dient als Statuszeile. Sie zeigt wichtige Informationen wie Fehlermeldungen etc. an. Normalerweise enthält sie in etwa den folgenden Text:

```
col: 8 line: 85 of 88 chgs: 205 file: Hello.mod
```

Dabei ist die Zahl hinter *col:* die Spalte (hier *8*) und die Zahl hinter *line:* die Zeile (hier *85*) in der sich die Schreibmarke (Cursor) gerade befindet. Danach folgt hinter *of* die Gesamtzahl der Zeilen des gerade bearbeiteten Textes. Die Zahl hinter *chgs:* gibt an, wieviele Änderungen (engl. changes) seit dem letzten Speichern am Text vorgenommen wurden. Zuletzt steht nach *file:* der Name der gerade bearbeiteten Datei (hier *Hello.mod*).

Während mit dem Kommando *startmacro* (siehe unten) ein Makro aufgezeichnet wird, steht am Ende der Statuszeile der Text *(recording)*.

Mit den Symbolen (Gadgets) am rechten Rand des OEd-Fensters kann man sich durch den Text bewegen. Das große Schieberegler-Symbol zeigt die relative Position des angezeigten Textausschnitts zum Ge-

samttext an. Mit ihm kann man sich schnell durch den Text bewegen. Durch Betätigung der Pfeil-Symbole unten rechts im Fenster kann man sich zeilenweise durch den Text bewegen. Durch langes Drücken dieser Symbole kann man sich langsam durch den Text bewegen.

Die Maus

Mit der Maus kann die Schreibmarke an eine bestimmte Position im Text gesetzt werden, indem man die gewünschte Position anfährt und kurz die linke Maustaste drückt.

Um mit der Maus einen Block zu markieren, muß man am Anfang des gewünschten Bereichs die Maustaste drücken und sie gedrückt lassen, bis man mit dem Mauszeiger über dem Ende des Bereichs steht. Dort kann die Taste losgelassen werden. Genaueres über die Bearbeitungsmöglichkeiten von Blöcken steht weiter unten im Abschnitt über 'Blöcke bearbeiten'.

Befehle von OEd

OEd hat keine feste Tastatur- oder Menübelegung mehr, sondern ist völlig frei programmierbar. Alle Funktionen können und müssen über ARexx-Befehle aufgerufen werden.

OEd arbeitet zwar auch auf Rechnern, auf denen ARexx nicht installiert ist, einige Funktionen können dort jedoch nicht angewendet werden. Da ARexx in AmigaOS 2.0 integriert wurde, wird Benutzern einer älteren Betriebssystemversion zum Umstieg auf AmigaOS 2.0 geraten.

Damit OEd ARexx benutzen kann, muß das Programm 'RexxMast' in der Datei 'S:User-Startup' gestartet werden. Genaueres dazu steht im Handbuch zu AmigaOS 2.0.

Eine Kurzbeschreibung der Kommandos ist in der Datei 'OEdRexx-

Help.txt' enthalten. Innerhalb von OEd wird diese Datei beim Drücken von 'HELP' automatisch in einem neuen Fenster angezeigt. Dazu muß sich 'OEdRexxHelp.txt' im Verzeichnis 'Oberon:' befinden.

Parameter, die an Kommandos übergeben werden, müssen in runde Klammern, Hochkommata oder Anführungszeichen eingeschlossen werden, wenn sie Leerzeichen enthalten. Innerhalb von Parametern kann der Schrägstrich rückwärts '\' dazu verwendet werden, die umschließenden Zeichen selbst zu benutzen. So kann z.B. mit 'write (:-\))' der Text ':-)' ausgegeben werden.

Die Kommandos im Einzelnen

In Hochkommata sind jeweils die ARexx-Kommandos angegeben. Darunter steht in Klammern die normalerweise verwendete Tastatur- und Menübelegung, wie sie in den mitgelieferten Konfigurationsdateien definiert ist.

Schalter

Schalter beeinflussen das Verhalten des Editors. Sie können alle entweder ein- oder ausgeschaltet sein.

Den Schalter-Kommandos kann allen ein Parameter übergeben werden. Dieser muß 'ON', 'OFF' oder 'TOGGLE' sein. Dabei bedeutet:

ON	**Der Schalter wird eingeschaltet**
OFF	**Der Schalter wird ausgeschaltet**
TOGGLE	**Der Schalter wird umgeschaltet**

Wird kein Parameter angegeben, so wird der Schalter umgeschaltet.

'insert [ON|OFF|TOGGLE]'
(AMIGA + ".", Menü 'Settings', Menüpunkt 'Insert')

Hiermit wird der Einfüge- (insert) oder der Überschreibmodus gewählt. Im Einfügemodus wird beim Schreiben der Text unter der Schreibmarke nach rechts verschoben. Im Überschreibmodus wird er gelöscht.

'inserton'
'insertoff'

Diese Kommandos sind lediglich für eine bessere Kompatibilität zu älteren Versionen von OEd. Es sind Synonyme für 'insert ON' bzw. 'insert OFF'.

'layout [ON|OFF|TOGGLE]'
(AMIGA + "-", Menü 'Settings', Menüpunkt 'Layout Mode')

Normalerweise befindet sich OEd im 'Layout-Modus'. Schaltet man diesen Modus ab, kann man die Schreibmarke auch in der ersten Spalte nach links und in der letzten Spalte einer Zeile nach rechts bewegen. Dabei wird die Schreibmarke an das Ende der vorigen Zeile bzw. an den Anfang der nächsten Zeile gesetzt.

'icons [ON|OFF|TOGGLE]'
(AMIGA + "0", Menü 'Settings', Menüpunkt 'Create Icons?')

Ist dieser Schalter aktiviert, werden beim Speichern der Texte Piktogramme erzeugt. Zudem wird diese Option beim Start des Compilers, Linkers der des Make-Utilities an diese Programme weitergegeben, so daß diese Programme auch Piktogramme erzeugen.

'script [ON|OFF|TOGGLE]'
(Menü 'Settings', Menüpunkt 'Set Script Flag?')

Dieser Schalter wird wichtig, wenn man mit OEd Amiga-

Befehlsdateien (Skriptdateien) schreiben möchte. Ist er aktiviert, wird beim Speichern des Textes das Skript-Flag gesetzt (siehe Amiga Handbuch).

Ist dieses Flag bei einer Datei gesetzt, dann wird es von OEd nicht mehr gelöscht, so daß man beim Ändern einer Skript-Datei diesen Schalter nicht explizit einzuschalten braucht.

Schreibmarke bewegen

'up'
'down'
'left'
'right' (Pfeiltasten)

Diese Kommandos bewegen die Schreibmarke um eine Position in die angegebene Richtung: hoch, runter, nach links bzw. nach rechts.

'top'
'bottom' (Umschalttaste + Pfeiltaste nach oben bzw. unten)

Die Schreibmarke wird in die erste bzw. in die letzte Zeile des Textes gesetzt.

'first'
'last' (Umschalttaste + Pfeiltaste nach links bzw. rechts)

Die Schreibmarke wird in die erste bzw. in die letzte Spalte der aktuellen Zeile gesetzt.

'wleft'
'wright' (Alt + Pfeiltaste nach links bzw. rechts)

Die Schreibmarke wird an den Anfang des Wortes links bzw. rechts neben der Schreibmarke gesetzt. Befindet sich das nächste Wort erst

in einer der nächsten Zeilen, wird die Schreibmarke in diese Zeile gesetzt.

'pageup'
'pagedown' (Alt + Pfeiltaste nach oben bzw. unten)

Die Schreibmarke wird um eine Bildschirmseite nach oben bzw. nach unten bewegt.

'scrollleft'
'scrollright'
'scrollup'
'scrolldown' (Steuerung + Pfeiltasten)

Der angezeigte Text wird so verschoben, daß sich die Schreibmarke relativ zu ihm um eine Position in die jeweilige Richtung bewegt. Die Schreibmarke selbst bleibt jedoch, wenn dies möglich ist, an der gleichen Position auf dem Bildschirm.

'tab'
'backtab' (Tabulatortaste, Umschalttaste + Tabulatortaste)

Die Schreibmarke wird an die nächste bzw. an die vorige Tabulatormarke gesetzt.

'ping <n>'
'pong <n>' (Umschalttaste + F1-F5, F1-F5)

OEd verfügt über 10 Textmarkierungen, die an beliebige Zeilen im Text gesetzt werden können. Die Markierungen sind von 1 bis 10 durchnummeriert. Mit 'ping <n>' wird die Markierung <*n*> auf die aktuelle Zeile gesetzt.

'pong <n>' setzt die Schreibmarke in die Zeile, auf die mit 'ping <n>' eine Markierung gesetzt wurde. Wurde die Markierung <*n*> noch

nicht gesetzt, setzt 'pong <n>' die Markierung *<n>* sie auf die aktuelle Zeile.

Wird eine Zeile gelöscht, auf die eine Markierung gesetzt wurde, so wird auch die Markierung gelöscht.

Bei der normalen Tastaturbelegung wird mit Shift + Funktionstaste eine der Markierungen 1 bis 5 gesetzt. Dieselbe Funktionstaste alleine gedrückt springt an die entsprechende Markierung.

Beispiel:

```
'ping 3'   /* Markierung 3 auf aktuelle Position */
'bottom'   /* an das Textende springen           */
'pong 3'   /* Zurück zur ursprünglichen Position */
```

'gotox <position>'

Die Schreibmarke wird in die Spalte *<position>* gesetzt. Ist *<position>* kleiner als eins, wird die Schreibmarke in die erste Spalte gesetzt.

Beispiel:

```
'gotox 30'  /* Schreibmarke in Spalte 30 setzen */
```

'gotoy [<position>]'

(Steuerung + "G", Menü 'Special', Menüpunkt 'Goto Line...')

Die Schreibmarke wird in die Zeile *<position>* gesetzt. Wird *<position>* nicht angegeben, oder ist die angegebene Zeilennummer kleiner als eins, wird ein Dialogfenster geöffnet, in das die gewünschte Zeilennummer eingegeben werden kann.

Ist die angegebene Zeilennummer größer als die Nummer der letzten Zeile, wird an das Textende gesprungen.

Beispiel:

```
'gotoy 100'; /* Schreibmarke nach Zeile 100 */
```

'mbracket'
(Steuerung + "H", Menü 'Special', Menüpunkt 'Matching Bracket')

Dieses Kommando ist nur sinnvoll, wenn sich die Schreibmarke auf einer Klammer (runde, eckige oder geschweifte) befindet. Die Schreibmarke wird bei einer öffnenden Klammer auf die entsprechende schließende und bei einer schließenden auf die entsprechende öffnende Klammer gesetzt.

'gotolastch'
(Steuerung + "A", Menü 'Special', Menüpunkt 'Goto Last Change')

Setzt die Schreibmarke an die Position der letzten Textänderung.

Blöcke bearbeiten

'bbegin' (Amiga + "B", Menü 'Block', Menüpunkt 'Begin')

Ist zur Zeit ein Block markiert, wird seine Markierung gelöscht. Der Anfang und das Ende der Markierung wird auf die aktuelle Zeile gesetzt.

'bend' (Amiga + "E", Menü 'Block', Menüpunkt 'End')

Wurde vor diesem Kommando der Anfang des Blocks mit 'bbegin' auf die aktuelle Zeile gesetzt, wird ein horizontaler Block in der aktuellen Zeile von der aktuellen Spalte beim Aufruf von 'bbegin' bis zur nun aktuellen Spalte markiert.

Ansonsten wird der markierte Block bis zur aktuellen Zeile verlängert.

'markall' (Steuerung + "K", Menü 'Block', Menüpunkt 'Mark All')

Der gesamte Text wird als Block markiert.

'bcopy' (Amiga + "Y", Menü 'Block', Menüpunkt 'Copy Block')

Der markierte Block wird an die aktuelle Position kopiert. Dies ist nur möglich, wenn sich die Schreibmarke außerhalb des markierten Blocks befindet.

'bmove' (Amiga + "M", Menü 'Block', Menüpunkt 'Move')

Der markierte Block wird an die aktuelle Position geschoben. Dies ist nur möglich, wenn sich die Schreibmarke außerhalb des markierten Blocks befindet.

'bdelete' (Amiga + "D", Menü 'Block', Menüpunkt 'Delete')

Der markierte Block wird gelöscht.

'cut' (Alt + "X", Menü 'Block', Menüpunkt 'Cut')

Der markierte Block wird gelöscht und in den Zwischenspeicher (des Clipboards) kopiert.

'copy' (Alt + "C", Menü 'Block', Menüpunkt 'Copy')

Der markierte Block wird in den Zwischenspeicher (des Clipboards) kopiert.

'paste' (Alt + "V", Menü 'Block', Menüpunkt 'Paste')

Der Inhalt des Zwischenspeichers (des Clipboards) wird an der aktuellen Position eingefügt.

'bunmark'
(Steuerung + "U", Menü 'Block', Menüpunkt 'Unmark')

Die Blockmarkierung wird gelöscht.

'btabright'
(Steuerung + "R", Menü 'Block', Menüpunkt 'TAB right')
'btableft'
(Steuerung + "L", Menü 'Block', Menüpunkt 'TAB left')

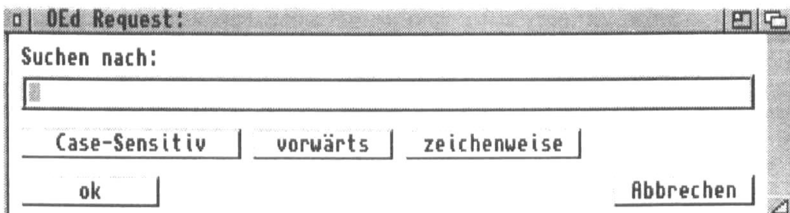
Der markierte Block wird um eine Tabulatorweite nach rechts bzw. nach links verschoben. Beim Schieben nach rechts werden links Leerzeichen eingefügt. Beim Schieben nach links gehen die Zeichen am linken Rand verloren.

Diese beiden Kommandos haben bei horizontalen Blöcken keine Funktion.

Suchen und Ersetzen

'find' (Amiga + "F", Menü 'Search', Menüpunkt 'Find...')

Es wird ein Dialogfenster geöffnet. Hier kann in einem großen Texteingabefeld eine Zeichenkette eingegeben werden. Diese wird im folgenden Suchtext genannt.



Über drei Schaltersymbole und über das Menü 'Settings' dieses Dialogfensters kann gewählt werden, auf welche Weise aufgrund des Suchtexts gesucht werden soll.

Mit dem ersten Schalter wird entschieden, ob bei der Suche Groß- und Kleinschreibung als gleich ('nicht Case-Sens.') oder als verschieden ('Case-Sensitiv') angesehen werden soll. Diesem Schalter entspricht der Menüpunkt 'Case-sensitive' und das Tastaturkürzel 'Amiga + "S"'.

Der mittlere Schalter wählt die Richtung, in der ausgehend von der aktuellen Position gesucht werden soll, 'vorwärts' oder 'rückwärts'. Diesem Schalter entspricht der Menüpunkt 'Forward' und das Tastaturkürzel 'Amiga + "F"'.

Mit dem dritten Schalter wird entschieden, ob nur nach ganzen Wörtern ('wortweise') oder auch nach Teilen von Wörtern und nach anderen Zeichen als Buchstaben ('zeichenweise') gesucht werden soll. Diesem Schalter entspricht der Menüpunkt 'Word by Word' und das Tastaturkürzel 'Amiga + "W"'.

Zwei Aktionssymbole und das Menü 'Project' dienen zum Verlassen des Dialogfenster:

Das Symbol 'ok', der Menüpunkt 'Ok' oder das Tastaturkürzel 'Amiga + "O"' startet die Suche nach dem Suchtext. Dies geschieht auch dann, wenn in das Texteingabefeld ein Suchtext eingegeben wurde und die Eingabetaste (Return) betätigt wird.

Wird der Suchtext gefunden, dann wird die Schreibmarke an den Anfang des gefundenen ersten Vorkommens gesetzt. Ansonsten blinkt der Bildschirm kurz auf.

Das Symbol 'Abbruch', der Menüpunkt 'Cancel' oder das Tastaturkürzel 'Amiga + "C"' verlassen das Dialogfenster, ohne die Suche zu starten. Dies geschieht auch dann, wenn in das Texteingabefeld kein Suchtext eingegeben wurde und die Eingabetaste (Return) betätigt wird.

'next' (Amiga + "N", Menü 'Search', Menüpunkt 'Next')

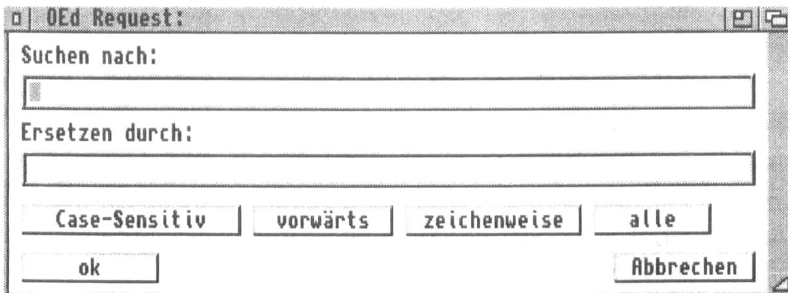
Das nächste Vorkommen des Suchtextes wird vorwärts im Text von der aktuellen Position ausgehend gesucht.

'prev' (Amiga + "P", Menü 'Search', Menüpunkt 'Previous')

Das nächste Vorkommen des Suchtextes wird rückwärts im Text von der aktuellen Position ausgehend gesucht.

'findrep'
(Amiga + "G", Menü 'Search', Menüpunkt 'Find-Replace...')

Es wird ein Dialogfenster geöffnet. Hier können in zwei großen Texteingabefeldern Zeichenketten eingegeben werden. Die obere Zeichenkette entspricht dem Suchtext bei 'find'. Mit der Eingabetaste kann vom oberen Texteingabefeld in das untere gewechselt werden. Der Text im unteren Texteingabefeld wird im folgenden Ersetztext genannt. Im Gegensatz zum Suchtext, kann der Ersetztext leer sein.



Über vier Schaltersymbole und über das Menü 'Settings' dieses Dialogfensters kann gewählt werden, auf welche Weise nach dem Suchtext gesucht und wie er durch den Ersetztext ersetzt werden soll.

Die ersten drei Schalter sind identisch mit denen des 'find' Dialogfensters (siehe oben).

Mit dem letzten Schaltersymbol, dem Menüpunkt 'Replace All' oder dem Tastaturkürzel 'Amiga + "A"' wird entschieden, ob nur das nächste Vorkommen ('einzeln') des Suchtextes ersetzt werden soll, oder ob alle Vorkommen ersetzt werde sollen ('alle').

Wie bei 'find' kann das Dialogfenster mit zwei Aktionssymbolen verlassen werden. Dabei wird das Symbol 'ok' automatisch gewählt, wenn im Texteingabefeld des Ersetzttextes die Eingabetaste gedrückt wird.

Wurde das Symbol 'ok' gewählt, sucht OEd nach dem Suchtext. Wurde der vierte Schalter auf 'einzeln' gestellt, so wird das nächste Vorkommen ersetzt, und das Suchen wird danach beendet.

Ist dagegen 'alle' angewählt, so wird die Schreibmarke an das erste Vorkommen des Suchtextes gesetzt und ein weiteres Dialogfenster wird geöffnet. Hier kann nun mit vier Schaltersymbolen oder der Tastatur gewählt werden, wie ersetzt werden soll:

Wird das Symbol 'ja!' angewählt oder die Taste 'J' gedrückt, wird das angezeigte Vorkommen des Suchtextes ersetzt und danach weiter gesucht.

Wird stattdessen das Symbol 'nein!' angewählt oder die Taste 'N' gedrückt, so wird weiter gesucht, ohne den Text zu ändern.

Mit dem Symbol 'alle' oder der Taste 'A' werden alle Vorkommen des Suchtextes durch den Ersetztext ersetzt.

Mit dem Symbol 'Beenden' oder der Taste 'B' wird das Suchen abgebrochen, ohne den Text zu verändern.

’nextrep’

(Amiga + "R", Menü ’Search’, Menüpunkt ’Next-Replace’)

Das nächste Vorkommen des Suchtextes wird vorwärts im Text gesucht und durch den Ersetztext ersetzt.

’prevrep’

(Amiga + "H", Menü ’Search’, Menüpunkt ’Previous-Replace’)

Das nächste Vorkommen des Suchtextes wird rückwärts im Text gesucht und durch den Ersetztext ersetzt.

’findstr <string>’

Der Suchtext wird auf den Text <*string*> gesetzt.

’repstr <string>’

Der Ersetztext wird auf den Text <*string*> gesetzt.

’casesens [ON|OFF|TOGGLE]’

(Menü ’Search’, Menüpunkt ’Case-sensitive’)

’forward [ON|OFF|TOGGLE]’

’wordbyword [ON|OFF|TOGGLE]’

(Menü ’Search’, Menüpunkt ’Word by Word’)

’all [ON|OFF|TOGGLE]’

Mit diesen vier Kommandos können die Einstellungen der Schaltersymbole der Dialogboxen bei ’find’ und ’findrep’ verändert werden.

Datei Ein-/Ausgabe-Kommandos

’load [<name>]’

(Amiga + "L", Menü ’Project’, Menüpunkt ’Load...’)

Die Datei <*name*> wird als neuer Text geladen. Wird <*name*> nicht

angegeben, wird ein Dateiauswahlfenster geöffnet, mit dem der Name eingegeben werden kann.

Der vorher bearbeitete Text wird beim Laden gelöscht. Wurde der Text verändert und nicht gespeichert, wird vorher noch gefragt, ob er gespeichert werden soll.

'save' (Amiga + "S", Menü 'Project', Menüpunkt 'Save')

Der Text wird unter seinem alten Namen gespeichert.

Existiert ein Text mit gleichem Namen bereits, wird dieser vor dem Speichern mit der Namensendung '.bak' versehen und nach dem Speichern gelöscht. Sollte durch irgendein Unglück der Computer während des Speicherns ausfallen, existiert wenigstens die alte Datei noch mit der Endung '.bak'.

Durch das Umbenennen der alten Datei kann es allerdings beim Speichern passieren, daß der Datenträger voll wird, obwohl noch genügend Platz für die veränderte Datei wäre. In diesem Fall hilft es, die Datei mit der Endung '.bak' mit dem Shell-Kommando 'delete' zu löschen. Dann nimmt man jedoch den vollständigen Verlust der Datei im Falle eines Computerausfalls in Kauf.

'saveas [<name>]'
(Amiga + "V", Menü 'Project', Menüpunkt 'Save as...')

Der Text wird unter dem angegebenen Namen wie bei *save* gespeichert. Wird kein Name angegeben, kann über ein Dateiauswahlfenster ein Name eingegeben werden. Existiert eine Datei mit dem dann gewählten Namen bereits, fragt OEd mit einem Dialogfenster noch einmal nach, ob die Datei überschrieben werden soll.

'bsave [<name>]'
(Amiga + "O", Menü 'Project', Menüpunkt 'Save Block...')

Der markierte Block wird unter dem angegebenen Namen gespeichert. Wird kein Name angegeben, wird ein Dateiauswahlfenster geöffnet. Existiert eine Datei mit dem dann gewählten Namen bereits, fragt OEd mit einem Dialogfenster noch einmal nach, ob die Datei überschrieben werden soll.

'insfile [<name>]'
(Amiga + "I", Menü 'Project', Menüpunkt 'Insert File...')

Fügt die angegebene Datei an über der aktuellen Position ein. Wird kein Name angegeben, so wird ein Dateiauswahlfenster geöffnet, mit dem der Name gewählt werden kann.

Druckkommandos

'print' (Steuerung + "P", Menü 'Project', Menüpunkt 'Print')

Der Text wird auf dem Drucker ausgegeben.

'printblock'
(Steuerung + "B", Menü 'Project', Menüpunkt 'Print Block')

Der markierte Block wird auf dem Drucker ausgegeben. Horizontale Blöcke können mit diesem Kommando nicht ausgedruckt werden.

'printas' (Steuerung + "I", Menü 'Project', Menüpunkt 'Print As...')

Es wird ein Dialogfenster geöffnet, mit dem sich die Einstellungen für das Drucken des Textes verändern lassen.

In einem großen Texteingabefeld kann der linke Rand des Textes eingestellt werden.

Über drei Schaltersymbole und das Menü 'Settings' kann die Schriftart gewählt werden.

Das erste Symbol wählt zwischen Briefqualität ('NLQ') und Entwurfsqualität ('Draft'). Diesem Symbol entspricht der Menüpunkt 'NLQ' und das Tastaturkürzel 'Amiga + "N"'.

Das mittlere Symbol wählt die Breite der Schrift: 'Normalschrift' oder 'Schmalschrift'. Dies kann auch mit dem Menüpunkt 'Condensed' oder dem Tastaturkürzel 'Amiga + "D" gewählt werden.

Mit dem letzten Symbol wird der Zeilenabstand festgelegt. Er kann entweder 6 ('6 LPI') oder 8 ('8 LPI') Zeilen pro Zoll betragen. Diesem Symbol entspricht der Menüpunkt '8 LPI' und das Tastaturkürzel 'Amiga + "8"'.

Zwei Aktionssymbole und das Menü 'Project' dienen zum Verlassen des Dialogfenster.

Das Symbol 'ok', der Menüpunkt 'Ok' oder das Tastaturkürzel 'Amiga + "O"' startet danach den Druck. Dies geschieht auch, wenn im Texteingabefeld die Eingabetaste (Return) betätigt wird.

Das Symbol 'Abbruch', der Menüpunkt 'Cancel' oder das Tastaturkürzel 'Amiga + "C"' verlassen das Dialogfenster, ohne das Drucken zu starten.

'printblkas'

(Steuerung + "O", Menü 'Project', Menüpunkt 'Print Block As...')

Es wird das gleiche Dialogfenster wie bei 'printas' geöffnet. Danach wird jedoch nicht der gesamte Text, sondern nur der markierte Block gedruckt. Horizontale Blöcke können mit diesem Kommando nicht ausgedruckt werden.

Kommandos zum Verändern der Fenster

'newwindow'
(Amiga + "W", Menü 'Project', Menüpunkt 'New Window...')

Es wird ein neues, leeres Textfenster geöffnet.

'quit'
(Amiga + "Q", Menü 'Project', Menüpunkt 'Quit')

Das aktive Fenster wird geschlossen. Wurde der Text in diesem Fenster verändert, wird vorher gefragt, ob er gespeichert werden soll.

Die anderen Textfenster bleiben weiterhin geöffnet und werden nicht verändert.

Ist das aktive Fenster das letzte geöffnete Fenster und existieren noch mit 'hide' (siehe unten) versteckte Texte, kann dieses Fenster nicht geschlossen werden. Es muß zunächst mindestens eines der versteckten Fenster geöffnet werden.

'iconify' (Escape)

Das aktive Fenster wird auf seine minimale Größe verkleinert und in den Hintergrund gebracht.

Nochmaliges Ausführen von 'iconify' vergrößert das Fenster wieder und bringt es in den Vordergrund.

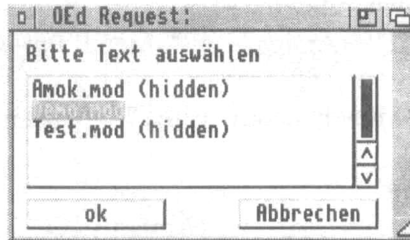
'hide' (Steuerung + Escape, Menü 'Project', Menüpunkt 'Hide')

Das aktive Fenster wird versteckt. Es kann mit 'reveal' wieder geöffnet werden.

Ist das aktive Fenster das einzige geöffnete Fenster, kann es nicht versteckt werden.

'reveal' (Steuerung + "V", Menü 'Project', Menüpunkt 'Reveal')

Es wird ein Dialogfenster mit allen im Augenblick bearbeiteten Texten angezeigt. Dabei sind alle mit 'hide' versteckten Texte mit *(hidden)* hinter ihrem Namen gekennzeichnet.



Mit der Maus oder mit den Pfeiltasten kann ein Text ausgewählt werden.

Wird das Symbol 'ok' oder der Menüpunkt 'Ok' angewählt, eine der Tastenkombinationen 'Amiga + "O"' oder 'Return' gedrückt oder ein Text mit der Maus doppelt angewählt, wird das Fenster, das diesen Text enthält, geöffnet und in den Vordergrund gebracht.

Mit dem Symbol 'Abbruch', dem Menüpunkt 'Cancel', der Tastenkombinationen 'Amiga + "C"' oder 'Escape' wird das Dialogfenster verlassen, ohne daß ein Fenster geöffnet oder in den Vordergrund gebracht wird.

'title <string>'

In der Titelzeile des aktiven Fensters wird der Text *<string>* angezeigt. Dieses Kommando ist sinnvoll, wenn man von einem ARexx-Programm aus Informationen an den Benutzer schicken möchte.

Beispiel:

```
'title' "'Freies RAM: " || storage()%1024 || "KB'"
   /* Der freie Systemspeicher wird angezeigt */
```

'shuffle' (Umschalttaste + Escape)

Das nächste Fenster aus der Liste der geöffneten Textfenster wird aktiviert und in den Vordergrund gebracht.

Kommandos zum Verändern des Textes

'write <string>'

Die Zeichenkette *<string>* wird an der aktuellen Position in den Text geschrieben.

Beispiel:

```
'write' "'" || date() || "'"
      /* Das Datum wird ausgegeben */
```

'writeasc <n>'

Das Zeichen mit dem ASCII-Wert *<n>* wird an der aktuellen Position in den Text geschrieben.

Beispiel:

```
/* Dieses kleine ARexx-Programm zeigt alle Zeichen
 * des Amiga-Zeichensatzes:
 */

address 'OEd'
'return'
'first'
do i= 32 to 127; 'writeasc' i; end

'return'
'first'
do i=160 to 255; 'writeasc' i; end
```

'return' (Return)

Unter der aktuellen Zeile wird eine neue Zeile eingefügt und die Schreibmarke unter das erste Zeichen der aktuellen Zeile gesetzt.

'split' (Umschalttaste + Return, Steuerung + "S")

Die aktuelle Zeile wird an der aktuellen Position in zwei Zeilen aufgeteilt. Am Anfang der zweiten Zeile werden so viele Leerzeichen eingefügt, wie die ursprüngliche Zeile an ihrem Anfang hatte. Die Schreibmarke wird auf das erste Zeichen der zweiten Zeile gesetzt.

'join' (Steuerung + "J")

Die folgende Zeile wird an diese Zeile angefügt. Zwischen beide Zeilen wird ein Leerzeichen eingefügt.

'back' (Rücktaste)

Das Zeichen links neben der Schreibmarke wird gelöscht. Befindet sich die Schreibmarke in der ersten Spalte, wird die aktuelle Zeile an die vorige Zeile wie mit *join* angehängt.

'del' (Löschtaste)

Das Zeichen unter der Schreibmarke wird gelöscht. Befindet sich die Schreibmarke am Ende einer Zeile, so wird die nächste Zeile wie bei *join* angehängt.

'deline'
(Umschalttaste + Rücktaste oder Umschalttaste + Löschtaste)

Die aktuelle Zeile wird gelöscht. Die gelöschte Zeile wird dabei in einen globalen Puffer kopiert, aus dem sie mit *undeline* wieder in den Text eingefügt werden kann. Dabei kann eine in einem Fenster gelöschte Zeile auch in einem anderen wieder eingefügt werden.

'undeline' (Alt + "C", Menü 'Block', Menüpunkt 'Copy')

Die zuvor mit *deline* gelöschte Zeile wird an der aktuellen Position eingefügt. Die gelöschte Zeile kann mit diesem Kommando auch mehrfach und in verschiedene OEd-Fenster, solange diese zum selben gestarteten OEd gehören, eingefügt werden.

'wback' (Alt + Rücktaste)
'wdel' (Alt + Löschtaste)

Alle Zeichen bis zum nächsten Wortanfang werden links von bzw. unter und rechts der Schreibmarke gelöscht.

'delbol' (Steuerung + Rücktaste)
'deleol' (Steuerung + Löschtaste)

Alle Zeichen der aktuellen Zeile links von bzw. unter und rechts der Schreibmarke werden gelöscht.

Änderungen rückgängig machen

'undo' (Steuerung + "<", Menü 'Special', Menüpunkt 'Undo')

Die letzte Textänderung wird rückgängig gemacht.

'undoall' (Menü 'Special', Menüpunkt 'Undo All')

Alle Textänderungen seit dem letzten Speichern oder Laden werden rückgängig gemacht. Wurden mehr Änderungen als die beim Start von OEd mit '-u' oder dem Tool Type MAXUNDOS angegebene maximale Anzahl gespeicherter Änderungen gemacht, werden nur diese rückgängig gemacht.

'redo' (Steuerung + ">", Menü 'Special', Menüpunkt 'Redo')

Die zuletzt rückgängig gemachte Textänderung wird nochmals

ausgeführt. Dieses Kommando kann nur ausgeführt werden, wenn zuvor eines der Kommandos 'undo' oder 'undoall' ausgeführt wurde.

'redoall' (Menü 'Special', Menüpunkt 'Redo All')

Alle mit 'undo' oder 'undoall' rückgängig gemachten Änderungen werden nochmals ausgeführt.

'clearundos'
(Steuerung + "Z", Menü 'Special', Menüpunkt 'Clear All Undos')

Der Speicher, der im Augenblick von den gemerkten Textänderungen belegt ist, wird freigegeben. Danach können die vor dem Aufruf von 'clearundos' gemachten Änderungen aber nicht mehr rückgängig gemacht werden.

Oberon-Unterstützung

'autouc [ON|OFF|TOGGLE]'
(Steuerung + ",", Menü 'Settings', Menüpunkt 'Auto Uppercase')

Ist dieser Schalter aktiviert werden Oberon-Schlüsselwörter und Standardbezeichner automatisch in Großschrift umgewandelt. Dadurch wird die Mechanik der Caps-Lock-Taste entlastet.

'mc68010 [ON|OFF|TOGGLE]'
'mc68020 [ON|OFF|TOGGLE]'
'mc68030 [ON|OFF|TOGGLE]'
(Menü 'Options', Menüpunkte '68010','68020','68030')

Mit diesen Schaltern wird der Prozessor gewählt, für den optimierter Code erzeugt werden soll.

Diese Schalter schließen sich gegenseitig aus, so daß beispielsweise 'mc68010 ON' automatisch die Schalter mc68020 und mc68030 ausschaltet.

'mc68881 [ON|OFF|TOGGLE]'
(Menü 'Options', Menüpunkt '68881/68882')

Mit diesem Schalter kann die Codeerzeugung für einen Fließkommaprozessor gewählt werden.

'debug [ON|OFF|TOGGLE]'
(Menü 'Options', Menüpunkt 'Debug')

Die Erzeugung von Code für den Debugger ODebug wird hiermit ein- bzw. ausgeschaltet.

'stackchk [ON|OFF|TOGGLE]'	(Amiga + "1")
'ovflchk [ON|OFF|TOGGLE]'	(Amiga + "2")
'rangechk [ON|OFF|TOGGLE]'	(Amiga + "3")
'casechk [ON|OFF|TOGGLE]'	(Amiga + "4")
'returnchk [ON|OFF|TOGGLE]'	(Amiga + "5")
'nilchk [ON|OFF|TOGGLE]'	(Amiga + "6")
'oddchk [ON|OFF|TOGGLE]'	
'typechk [ON|OFF|TOGGLE]'	(Amiga + "7")
'clearvars [ON|OFF|TOGGLE]'	
'autoregpars [ON|OFF|TOGGLE]'	
'clearvars [ON|OFF|TOGGLE]'	
'newsymfile [ON|OFF|TOGGLE]'	
'garbagecollector [ON|OFF|TOGGLE]'	(Amiga + "+")
'extensions [ON|OFF|TOGGLE]'	(Amiga + "#")
	(Menü 'Options')

Mit diesen Optionen können die verschiedenen Compilieroptionen gesetzt bzw. gelöscht, wie zum Beispiel die Optionen zur Auswahl des Überprüfungscodes, den der Compiler erzeugen soll.

'smallcode [ON|OFF|TOGGLE]'
(Amiga + "8", Menü 'Options', Menüpunkt 'Small Code')
'smalldata [ON|OFF|TOGGLE]'
(Amiga + "9", Menü 'Options', Menüpunkt 'Small Data')

Diese Schalter wählen das Speichermodell, welches der Oberon-Compiler und der Linker verwenden sollen.

'parse' (Amiga + "A", Menü 'Oberon', Menüpunkt 'Parse')

Es wird geprüft, ob der gerade bearbeitete Text der Oberon-Syntax entspricht.

'compile' (Amiga + "C", Menü 'Oberon', Menüpunkt 'Compile...')

Der Oberon-Compiler wird auf dem gerade bearbeiteten Text gestartet. Der Compiler übernimmt den Text dabei direkt aus dem Speicher, so daß er vor der Compilation nicht gespeichert werden muß.

'link' (Amiga + "K", Menü 'Oberon', Menüpunkt 'Link...')

Der Linker OLink wird so gestartet, daß er das zuvor mit 'compile' compilierte Programm linkt.

'make' (Steuerung + "M", Menü 'Oberon', Menüpunkt 'Make...')

Das Make-Utility OMake wird mit dem gerade bearbeiteten Text als Argument gestartet. Dazu muß der Text, wenn er verändert wurde, zuerst gespeichert werden.

'execute' (Amiga + "X", Menü 'Oberon', Menüpunkt 'Execute...')

Das zuvor mit 'link' gelinkte Programm wird gestartet. Dabei wird ein Start von der Workbench ohne Argumente simuliert.

'coptions [<options>]'
(Steuerung + "T", Menü 'Oberon', Menüpunkt 'Compiler Options...')

Die Compileroptionen für bedingte Compilation (z.B. "SET English") werden auf *<options>* gesetzt. Wird *<options>* nicht angegeben, wird ein Dialogfenster geöffnet, mit dem die Optionen eingegeben werden können.

Im Dialogfenster können die Optionen in ein Texteingabefeld eingegeben werden. Die Optionen werden mit dem Symbol 'ok' übernommen. Durch Anwählen von 'Abbruch' bleiben sie unverändert.

'firsterror'
(Amiga + "Z", Menü 'Oberon', Menüpunkt 'First Error')

Traten bei der Compilation des Textes mit 'compile' Fehler auf, springt 'firsterror' an die Position des ersten Fehlers und zeigt die Fehlermeldung in der Titelzeile an.

'nexterror'
(Amiga + "T", Menü 'Oberon', Menüpunkt 'Next Error')

Traten bei der Compilation des Textes mit 'compile' mehrere Fehler auf, springt 'nexterror' an die Position des von der aktuellen Position aus nächsten Fehlers und zeigt die Fehlermeldung in der Titelzeile an.

'reloaderrs'
(Steuerung + "Q", Menü 'Oberon', Menüpunkt 'Reload Errorfile')

Mit dieser Option wird die vom Compiler erzeugte Fehlerdatei neu geladen. Dies ist nötig, wenn sich diese Datei geändert hat, ohne daß OEd dies bemerken konnte, also z.B. wenn man den Text nicht mit 'compile', sondern von der Workbench aus compiliert hat.

ARexx-Kommandos mit Ergebnis

Diese Kommandos sind nur beim Aufruf innerhalb von ARexx-Programmen sinnvoll.

Damit sie verwendet werden können, müssen im ARexx-Programm Ergebnisse mit 'options results' aktiviert werden. Die Ergebnisse können dann jeweils aus der Variablen 'result' ausgelesen werden.

'getch'

Ergibt das Zeichen an der aktuellen Position.

Beispiel:

```
/*
 * Zeichen an aktueller Position in Großbuchstabe
 * umwandeln:
 */
address 'OEd'
options results

insert off
getch
write upper(result)
insert on
```

'getasc'

Ergibt den ASCII-Wert des Zeichens an der aktuellen Position. Befindet sich unter der Schreibmarke kein Zeichen, da sich die Schreibmarke hinter dem Ende einer Zeile befindet, ist das Ergebnis 0.

Beispiel:

```
/* ASCII-Wert des aktuellen Zeichens anzeigen: */

  address 'OEd'
  options results

  getasc
  title "'ASCII-Wert = " || result ||"'"
```

'getline'

Der Inhalt der aktuellen Zeile wird zurückgegeben.

'getposx'
'getposy'

Die aktuelle horizontale bzw. vertikale Position wird zurückgegeben.

'numlines'

Ergibt die Anzahl der Zeilen des Textes.

Beispiel:

```
/* Zeilen durchnummerieren: */

address 'OEd'
options results

top
numlines
do i=1 to result
  first
  write i || ":"
  down
end
```

'getword'

Das Ergebnis ist das Wort, auf dem sich die Schreibmarke befindet. Dabei werden ausgehend von der Position der Schreibmarke alle zusammenhängenden alphanumerischen Zeichen vor, unter und hinter der Schreibmarke zurückgegeben.

'blockbegin'
'blockend'

Das Ergebnis ist die Nummer der ersten bzw. letzten Zeile des markierten Blocks. Ist kein Block markiert oder ist der markierte Block ein horizontaler Block, so ist das Ergebnis jeweils Null.

ARexx-Unterstützung

Die folgenden Kommandos ermöglichen das Arbeiten mit ARexx-Programmen. Sie können daher nur verwendet werden, wenn ARexx installiert ist.

'stoprexx'
(Steuerung + "C", Menü 'Special', Menüpunkt 'Stop Rexx (HI)')

Alle ARexx-Programme werden abgebrochen. Dazu wird das Programm 'HI' gestartet. 'HI' muß sich dazu im Verzeichnis 'REXXC:', 'SYS:REXXC' oder einem aktuellen Pfad befinden.

'enterrexx'
(Steuerung + "E", Menü 'Special', Menüpunkt 'Enter Rexx Cmd...')

Ein Dialogfenster wird geöffnet, in das eine kurze ARexx-Befehlsfolge eingegeben werden kann. OEd hängt automatisch die Anweisung 'address "OEd"' vor die Folge, so daß direkt auf die Kommandos von OEd zugegriffen werden kann.

Zwei Aktionssymbole und das Menü 'Project' dienen zum Verlassen des Dialogfenster.

Das Symbol 'ok', der Menüpunkt 'Ok', das Tastaturkürzel 'Amiga + "O"' oder die Eingabetaste im Texteingabefeld führt danach die Befehlsfolge aus. Die Kommandos werden asynchron ausgeführt, man kann also längere Befehlsfolgen 'stören', indem man im OEd weitertippt. Speziell kann man von OEd aus mit 'Steuerung + "C"' die gerade gestartete Befehlsfolge abbrechen. Dies ist vor allem dann nötig, wenn man durch einen Tipp- oder Denkfehler eine Befehlsfolge geschrieben hat, die den Text zerstört oder gar endlos läuft, ohne das Gewünschte zu tun.

Das Symbol 'Abbruch', der Menüpunkt 'Cancel' oder das Tastaturkürzel 'Amiga + "C"' verlassen das Dialogfenster, ohne die Befehlsfolge auszuführen.

'execrexx [<commands>]'
(Steuerung + "X", Menü 'Special', Menüpunkt 'Execute Rexx Cmd')

Die Zeichenkette *<commands>* wird als ARexx-Befehlsfolge wie bei *enterrexx* ausgeführt. Wird *<commands>* nicht angegeben, wird die zuletzt bei *enterrexx* eingegebene Befehlsfolge ausgeführt.

'rx <file>' (F6-F10, Menü 'Macros', Menüpunkt 'Play Macro', Alt oder Steuerung + F1-F10)

Das ARexx-Programm mit dem Namen *<file>* wird asynchron gestartet.

Mit den Tasten 'F6' bis 'F10' werden die ARexx-Programme 'T:RexxMacro_F6.oed' bis 'T:RexxMacro_F10.oed' ausgeführt (siehe *startmacro* weiter unten).

Die Funktionstasten sind so belegt, daß sie zusammen mit den Tasten Alt oder Steuerung gedrückt die ARexx-Programme 'aF#.oed',

'cF#.oed' oder 'acF#.oed' aus dem Verzeichnis 'Oberon:Rexx/' ausführen. 'F#' steht dabei für die jeweils gedrückte Funktionstaste.

'startmacro [<file>]' (Umschalttaste + F6-F10, Menü 'Macros', Menüpunkt 'Start Learning')

Alle nach diesem Befehl von OEd ausgeführten Kommandos werden als ARexx-Programm in die Datei *<file>* gespeichert. Wird *<file>* nicht angegeben, wird als Name "T:Macro.oed" verwendet. Die erzeugte Datei kann danach wie ein gewöhnliches ARexx-Programm ausgeführt werden.

Während OEd ein Makro aufzeichnet, hat *startmacro* die gleiche Funktion wie *stopmacro*.

Bei der mitgelieferten Tastaturbelegung werden mit den Tastenkombinationen 'Umschalttaste + "F6" bis "F10"' die ARexx-Programme 'T:RexxMacro_F6.oed' bis 'T:RexxMacro_F10.oed' aufgezeichnet.

Diese werden durch die entsprechende Funktionstaste alleine gedrückt mit dem Kommando *rx* ausgeführt (siehe oben).

'stopmacro' (Menü 'Macros', Menüpunkt 'Stop Learning')

Beendet das mit *startmacro* begonnene Aufzeichnen eines ARexx-Programms.

Tastatur- und Menüeinstellungen

'key <key>'

Das Kommando, mit dem die Taste *<key>* belegt ist, wird ausgeführt. Siehe unten, 'Tastaturbelegung'.

Beispiel:

```
'key (ESC CONTROL)'
```

'map <key> <mapping>'

Belegt die Taste *<key>* mit der Belegung *<mapping>*. Siehe unten, 'Tastaturbelegung'.

Beispiel:

```
'map (RETURN CONTROL (split))'
```

'loadkeys [<file>]'

Die Tastaturbelegung wird aus der Datei *<file>* geladen. Wird *<file>* nicht angegeben, so wird die Tastaturbelegung aus der Datei "Oberon:OEd_Keys.txt" geladen. Siehe unten, 'Tastaturbelegung'.

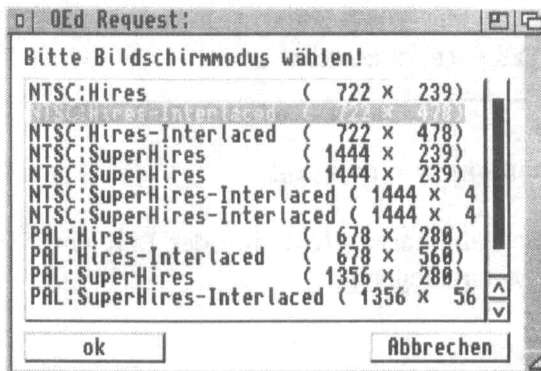
'loadmenu [<file>]'

Die Menübelegung wird aus der Datei *<file>* geladen. Wird *<file>* nicht angegeben, so wird die Menübelegung aus der Datei "Oberon:OEd_Menu.txt" geladen. Siehe unten, 'Menübelegung'.

Voreinstellungen

'screenmode'
(Steuerung + "D", Menü 'Project', Menüpunkt 'Screen Mode...')

Dieses Kommando funktioniert nur in Zusammenhang mit AmigaOS ab Version 2.04. Es wird ein Dialogfenster geöffnet, mit dem der Bildschirmmodus gewählt werden kann.



Mit der Maus oder mit den Pfeiltasten kann ein Bildschirmmodus ausgewählt werden.

Wird das Symbol 'ok' oder der Menüpunkt 'Ok' angewählt, eine der Tastenkombinationen 'Amiga + "O"' oder 'Return' gedrückt oder ein Bildschirmmodus mit der Maus doppelt angewählt, wird die Information über den gewählten Bildschirmmodus in der Datei 'Oberon:OEd_ScreenMode' gespeichert. Beim nächsten Start von OEd auf einem eigenen Screen wird dieser Modus verwendet.

Mit dem Symbol 'Abbruch', dem Menüpunkt 'Cancel' oder einer der Tastenkombinationen 'Amiga + "C"' oder 'Escape' wird das Dialogfenster verlassen, ohne daß der voreingestellte Bildschirmmodus verändert wird.

'tabwidth <n>'

Die Tabulatorweite wird auf <*n*> Zeichen gesetzt.

Sonstige Kommandos

'about' (Amiga + "J", Menü 'Project', Menüpunkt 'About...')

Es werden Informationen über die Versionsnummer von OEd und das Kopierrecht in einem Dialogfenster angezeigt. Das Symbol 'ok' schließt dieses Fenster wieder.

Tastaturbelegung

Die Tastaturbelegung von OEd wird in der Datei 'Oberon:OEd_Keys.txt' beschrieben. Fehlt diese Datei, ist der Editor völlig unbrauchbar, da allen Tasten von vornherein unbelegt sind, bis auf diejenigen, die Text ausgeben.

In 'Oberon:OEd_Keys.txt' wird jede Tastenbelegung in jeweils einer Zeile beschrieben. Kommentare beginnen mit einem Strichpunkt. Eine Tastenbelegung beginnt immer mit dem Namen der Taste, die belegt werden soll. Dies ist entweder das Zeichen, das die Taste gewöhnlich auf dem Bildschirm ausgibt, wenn sie gedrückt wird, oder bei Sondertasten einer der folgenden Namen:

RETURN	Eingabetaste
ENTER	Eingabetaste des Zehnerblocks
LEFT	Pfeiltaste links
RIGHT	Pfeiltaste rechts
UP	Pfeiltaste hoch
DOWN	Pfeiltaste hinunter
DEL	Löschtaste
BS	Rücktaste
HELP	Hilfe-Taste
TAB	Tabulatortaste
ESC	Escapetaste
SPACE	Leertaste
F1 bis F10	Funktionstasten
NK0 bis NK9	Nummerntasten des Zehnerblocks

Nach dem Namen der Taste folgt eine, möglicherweise leere, Liste von Umschalttasten, mit denen die Taste zusammen gedrückt werden soll.

Erlaubt ist hier eine beliebige Kombination der folgenden Umschalttasten:

```
SHIFT          Umschalttaste
CAPSLOCK       Feststelltaste
COMMAND        linke Amiga-Taste
AMIGA          rechte Amiga-Taste
ALT            Alt-Taste
CONTROL        Steuerungs-Taste
```

Bei der Belegung von Tastenkombinationen mit der Umschalttaste ist es gleich, ob die Taste in ihrem umgeschalteten Zustand oder zusammen mit 'SHIFT' angegeben wird. Die Belegung der Tastenkombination *a SHIFT* ist also gleichbedeutend mit der Belegung von *A*. Werden *a SHIFT* und *A* mit unterschiedlichen Kommandos belegt, wird die Belegung bei *a SHIFT* verwendet und jene von *A* ignoriert.

Nach der Liste der Umschalttasten folgt nun das Kommando, mit dem die Taste belegt sein soll. Alle Tasten können nur mit einem einzigen OEd-Kommando belegt werden. Dieses Kommando und der Parameter des Kommandos, wenn vorhanden, müssen in runden Klammern eingeschlossen sein.

Um eine Taste mit mehreren Kommandos zu belegen, muß sie mit dem Kommando *execrexx* belegt werden, wobei der Parameter von *execrexx* die Liste der Kommandos enthält.

Beispiele:

```
RETURN                  (return)
RETURN SHIFT            (split)
RETURN CONTROL          (execrexx ('return'; 'first'))
ENTER                   (key "RETURN SHIFT")
) ALT                   (write (:-\\\)))
9 ALT SHIFT             (write ":-\)")
```

Die Eingabetaste wird hier einfach mit *return* belegt. Zusammen mit der Umschalttaste wird sie mit *split* belegt.

Die Eingabetaste zusammen mit der Steuerungstaste gedrückt startet ein kleines ARexx-Programm, das eine Zeile unter der aktuellen Zeile einfügt und die Schreibmarke an den Anfang dieser Zeile setzt. Dies funktioniert natürlich nur, wenn ARexx korrekt installiert ist.

Die Eingabetaste des Nummernblocks wird so belegt, daß sie sich immer gleich verhält wie die Eingabetaste mit der Umschalttaste gedrückt. So wird die Belegung von 'ENTER' auch immer dann geändert, wenn 'RETURN SHIFT' neu belegt wird.

Die schließende runde Klammer und Alt gibt bei dieser Belegung den Text ':-)' aus. Es sind hier drei Schrägstriche rückwärts nötig, da der Text ':-)' doppelt in Klammern eingeschlossen ist. Der erste Schrägstrich rückwärts schützt beim Entfernen der äußeren Klammern den zweiten und der dritte die erste schließende Klammer. Als Tastenbelegung bleibt also 'write (:-\))' übrig. Der übrige Schrägstrich rückwärts schützt nun wie zuvor der dritte die Klammer nach ihm, so daß bei der Ausgabe der Text ':-)' übrig bleibt.

Eine einfachere, gleichwertige Belegung ist die von '9 ALT SHIFT'.

Die Zeilen in der Tastaturbelegungsdatei dürfen nicht länger als 256 Zeichen sein.

Menübelegung

Die Menübelegung von OEd wird in der Datei 'Oberon:OEd_Menu.txt' beschrieben. Ist diese Datei nicht vorhanden, besitzt OEd keine Menüs. Wie die Tastaturbelegung darf auch diese Datei Kommentare enthalten. Sie müssen mit einem Strichpunkt beginnen.

Für jede Menüleiste steht in dieser Datei ein Textblock, der mit der Zeile

```
MENU <Name>
```

beginnt. *<Name>* ist dabei der Name, den der folgende Menüstreifen bekommen soll. Ein Menüstreifen besteht aus anwählbaren Menüpunkten und Trennstrichen. Ein Menüpunkt wird mit

```
ITEM <Name> <Kommando>
```

definiert. Dabei ist *<Name>* der Name, den dieser Menüpunkt bekommen soll. *<Kommando>* ist wie bei der Tastaturbelegung das Kommando, mit dem dieser Menüpunkt belegt werden soll. Das Kommando und sein Parameter müssen in runde Klammern eingeschlossen sein. Jeder Menüpunkt kann nur mit einem Kommando belegt werden. Um einen Menüpunkt mit mehreren Kommandos zu belegen, muß er mit dem Kommando *execrexx* belegt sein, wobei der Parameter die Liste der Kommandos enthält.

Ein Trennstrich wird im Menü durch eine Zeile mit dem Text

```
SEPARATOR
```

eingefügt. Logisch zueinander gehörende Menüpunkte sollten mit Trennstrichen von anderen Menüpunkten getrennt werden.

In Menüpunkte, deren Kommando mit dem gleichen Parameter auch als Tastaturbelegung in 'Oberon:OEd_Keys.txt' definiert ist, erscheint bei der Anzeige automatisch rechts die entsprechende Taste. Um Tastaturabkürzungen für die Menüpunkte zu definieren reicht es also, die gewünschte Taste, wie oben beschrieben, mit dem gleichen Kommando zu belegen.

Beispiel:

```
MENU "Projekt"

ITEM "Laden..."   (load)
ITEM "Speichern"  (save)
SEPARATOR
ITEM "Löschen"    (execrexx ('markall'; 'bdelete'))
SEPARATOR
ITEM "Beenden"    (quit)

MENU "Einstellungen"

ITEM "Einfügemodus?"             (insert)
ITEM "Layout Modus?"             (layout)
SEPARATOR
ITEM "Piktogramme erzeugen?"  (icons)
ITEM "Skript-Flag setzen?"    (script)
```

Das von dieser einfachen Belegung erzeugte Menü sieht folgendermaßen aus:

Projekt
Laden...
Speichem
Löschen
Beenden

Einstellungen
✓ Einfügemodus?
✓ Layout Modus?
Piktogramme erzeugen?
Skript-Flag setzen?

Existiert noch eine Tasturbelegungsdatei 'Oberon:OEd_Keys.txt' mit diesen Kommandos, dann würden die Menüs auch die entsprechenden Tastaturkürzel enthalten.

Mitgelieferte ARexx-Makros

Um das Arbeiten mit OEd und das Programmieren in Oberon zu erleichtern, werden bereits kleine ARexx-Makros mitgeliefert. Diese können natürlich nur dann benutzt werden, wenn ARexx installiert ist. Im folgenden wird die voreingestellte Tastatur- und Menübelegung angegeben und das dazugehörende Makro erklärt:

Steuerung + Return

Dieses Makro führt ein *return* aus, setzt danach die Schreibmarke jedoch an den Anfang der neuen Zeile und nicht unter des erste Zeichen der darüberliegenden.

Help

Mit dieser Taste wird ein neues OEd-Fenster geöffnet in dem der Text 'Oberon:OEdRexxHelp.txt' angezeigt wird. Hier findet man eine kurze Beschreibung aller ARexx-Kommandos von OEd.

Alt + "F", Menü 'Search', Menüpunkt 'Find Word'

Oft ist es nötig, nach weiteren Vorkommen eines Wortes zu suchen, das sich bereits auf dem Bildschirm befindet. Dieses Makro verwendet daher das Wort unter der Schreibmarke als Suchtext und sucht nach dem nächsten Vorkommen dieses Wortes. Die Schreibmarke wird auf das gefundene Vorkommen gesetzt.

Steuerung + "ß", Menü 'Settings', Menüpunkt 'No Umlauts'

Die Umlauttasten und die "ß"-Taste werden mit den Zeichenkombinationen 'ae', 'oe', etc. und 'ss' belegt. So kann man mit OEd Texte ohne Umlaute schreiben, ohne daß man den Schreibmaschinenkurs vergessen muß. Dies ist z.B. dann sinnvoll, wenn man mit OEd elektronische Nachrichten (EMail) schreiben möchte, die gewöhnlich keine deutschen Sonderzeichen enthalten dürfen.

Umschalttaste + Steuerung + 'ß'
Menü 'Settings', Menüpunkt 'Umlauts'

Dieses Makro belegt die Umlauttasten und die "ß"-Taste wieder mit den gewöhnlichen Zeichen.

Steuerung + F1, Steuerung + F2

Der Markierte Block wird in die Datei 'T:blk' gespeichert bzw. aus dieser Datei gelesen und in den Text eingefügt. Dabei wird einem das manchmal lästige Dateiauswahlfenster bei den Kommandos *bsave* und *insertfile* erspart.

Steuerung + F3

Dieses Makro setzt die Schreibmarke um zehn Zeilen tiefer und danach wieder an die ursprüngliche Position. Dies ist bei längeren Texten sinnvoll, wenn sich die Schreibmarke um unteren Fensterrand befindet und man den Text unter der aktuellen Zeile sehen möchte, die Position der Schreibmarke jedoch unverändert bleiben soll.

Steuerung + F6, Steuerung + F7

Steuerung + F6 setzen den markierten Block in Oberon-Kommentare. So kann z.B. leicht eine Oberon-Prozedur 'auskommentiert' werden, die im Augenblick nicht benötigt wird, deren Text jedoch noch nicht gelöscht werden soll.

Steuerung + F7 macht aus einem so auskommentierten Block wieder den ursprünglichen Block. Dazu muß der auskommentierte Bereich als Block markiert sein.

Steuerung + F10

Dieses Makro speichert den Text im aktiven Fenster und schließt da-

nach das Fenster. Dies entspricht einer Save&Quit Funktion, wie sie andere Editoren kennen.

Alt + F1

Es wird ein 'leeres' Oberon-Modul erzeugt, das lediglich aus *MODULE*, *BEGIN* und *END* besteht. Der Name des Moduls und der eigentliche Text muß noch eingefügt werden.

Alt + F2

Es wird eine IMPORT-Anweisung erzeugt, in die gleich die Namen der importierten Module eingefügt werden kann.

Alt + F3, Alt + F4, Alt + F5

Die Schlüsselwörter *VAR*, *TYPE* bzw. *CONST* werden in den Text eingefügt. Danach können dann gleich die entsprechenden Deklarationen in den Text geschrieben werden.

Alt + F6

Eine leere Prozedur wird erzeugt, die noch um den Namen und die Parameter, den Deklarationsteil und die Anweisungen ergänzt werden muß.

Alt + F7, Alt + F8

Diese Makros erzeugen leere WHILE- und REPEAT-Schleifen, die jeweils noch um die Abbruchsbedingung und die Schleifenanweisungen erweitert werden müssen.

Alt + F9, Alt + F10

Diese Makros erzeugen eine leere IF- bzw. eine leere CASE-

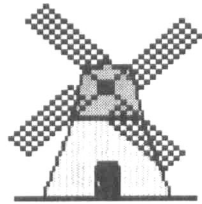
Anweisung, die noch entsprechend um Bedingungen bzw. Ausdrücke und Anweisungen ergänzt werden müssen.

Einschränkungen

Aus Effizienzgründen und durch Begrenzungen der Hardware unterliegt OEd verschiedenen Einschränkungen, die sich jedoch beim Arbeiten mit dem Editor praktisch nicht restriktiv auswirken dürften. Die Einschränkungen im Einzelnen:

Zeilenlänge:	max. 2147483647 Zeichen
Textlänge:	max. 2147483647 Zeilen
Suchtext:	max. 255 Zeichen
Ersetztext:	max. 255 Zeichen
Länge einer Tastenbelegung:	max. 255 Zeichen
Länge einer Menübelegung:	max. 255 Zeichen
Länge der ARexx-Parameter:	max. 255 Zeichen
Länge der ARexx-Resultate:	max. 255 Zeichen
Länge des ARexx-Kommandos bei *enterrexx*:	max. 255 Zeichen
Länge der Compileroptionen bei *coptions*:	max. 255 Zeichen
Tabulatorweite:	max. 2147483647
Anzahl gleichzeitig bearbeiteter Texte:	beliebig
Größe des Undo-Puffers (Option '-u'):	max. 2147483647 Einträge

5. Der Compiler Oberon



Das Programm 'Oberon' ist der eigentliche Compiler. Er übernimmt den größten Teil der Arbeit, der zum Übersetzen von Oberon-Quelltexten [Reiser 92] in lauffähige Amiga-Programme nötig ist.

Der Compiler erhält die mit OEd oder einem anderen Texteditor geschriebenen Quelltexte als Eingabe. Daraus erzeugt er verschiedene Dateien. Die wichtigste ist die Objektdatei: Sie enthält den Maschinenbefehlscode, aus dem das lauffähige Programm aufgebaut ist. Außerdem wird eine Symboldatei erzeugt. Diese enthält eine für den Compiler einfach zu lesende Zusammenfassung der nach außen sichtbaren (exportierten) Bezeichner des übersetzten Modul. Beim Übersetzen eines Moduls benötigt der Compiler alle Symboldateien der von diesem Modul benutzten (importierten) Module.

Enthält der übersetzte Quelltext Fehler, erzeugt der Compiler weder eine Objekt- noch eine Symboldatei, sondern lediglich eine Fehlerdatei. Diese Datei kann vom Texteditor oder von anderen Werkzeugen wie OErr (siehe Kapitel 6) benutzt werden, um die fehlerhaften Stellen im Quelltext anzuzeigen.

Die Namen der verwendeten Dateien

Um die verschiedenen Arten von Dateien, die beim Programmieren in Oberon vorkommen, leichter unterscheiden zu können, bekommen ihre Namen unterschiedliche Endungen. Die Endungen werden durch einen Punkt von dem eigentlichen Namen der Datei getrennt. So bekommt der Quelltext eines Moduls mit dem Namen 'Test' den Dateinamen 'Test.mod'. Die von Amiga Oberon verwendeten Endungen im Einzelnen sind:

mod	Quelltexte von Oberon-Modulen.
modE	Vom Compiler beim Übersetzen eines fehlerhaften Quelltextes erzeugte Fehlerdatei. Diese Datei kann mit dem Programm OErr (Kapitel 7) oder einem Editor, beispielsweise mit OEd (Kapitel 3 und 4), betrachtet werden.
def	Mit dem Werkzeug 'ModToDef' erzeugtes Definitionsmodul (Kapitel 7).
obj	Vom Compiler erzeugte Objektdatei.
objs	Vom Compiler erzeugte Objektdatei bei Verwendung des kleinen Datenmodells (Kapitel 14, Speichermodelle).
obja	Vom Compiler erzeugte Objektdatei. Bei der Compilation wurde der Garbage-Collector deaktiviert (Kapitel 16).
objsa	Vom Compiler erzeugte Objektdatei bei Verwendung des kleinen Datenmodells und deaktiviertem Garbage-Collector.
sym	Symboldatei, vom Compiler erzeugt.
ref	Vom Compiler erzeugte Referenzdatei. Sie wird vom Debugger 'ODebug' (als Zusatzpaket erhältlich) benötigt.
dis	Disassemblerlisting einer Objektdatei oder eines Programms. Diese Dateien werden vom Werkzeug 'DecObj' (Teil des ODebug-Pakets) erzeugt.
xref	Von 'XRef' (Teil des ODebug-Pakets) erzeugte Querverweisliste.
-	Ausführbares Programm nach dem Linken mit OLink (Kapitel 6).

Das Projekt-Konzept

Beim Entwickeln größerer Programme, die aus vielen Modulen bestehen, kann die Vielzahl an verschiedenen Dateien ein Verzeichnis schnell sehr unübersichtlich gestalten. Um eine bessere Übersicht über

ein Programmprojekt zu behalten, empfiehlt es sich für die verschiedenen Arten von Dateien Unterverzeichnisse zu erzeugen. Amiga Oberon unterstützt dabei die Unterverzeichnisse 'txt', 'sym', 'obj', 'bin' und 'ref'. Bei einem größeren Projekt sollten Sie diese Verzeichnisse mit dem Shell-Befehl 'makedir' erzeugen. Von der Workbench aus können Sie diese Verzeichnisse mit 'neue Schublade' des 'Fenster'-Menüs erzeugen. Die Unterverzeichnisse enthalten dann jeweils folgende Dateien:

txt	Quelltexte, Definitionsmodule, Disassemblerlistings und Kreuzreferenzen
sym	Symboldateien
obj	Alle Arten von Objektdateien
bin	Ausführbare Programme
ref	Referenzdateien

Existieren solche Unterverzeichnisse, so bestehen die Programme des Amiga Oberon Systems darauf, daß die entsprechenden Dateien auch dort abgelegt sind. Die automatisch erzeugten Dateien, wie z.B. die Symbol- und Objektdateien, werden auch automatisch in die entsprechenden Verzeichnisse geschrieben.

Es ist natürlich möglich, nur einzelne dieser Verzeichnisse anzulegen, andere jedoch nicht. So ist es beispielsweise oft nicht sinnvoll, für die ausführbaren Programme das Unterverzeichnis 'bin' zu verwenden, insbesondere wenn man nur an einem einzigen ausführbaren Programm arbeitet.

Aufruf des Compilers

Aufruf aus dem Editor

Schreiben Sie Ihre Quelltexte mit dem Editor OEd, so ist der Aufruf des Compilers aus dem Editor selbst wohl am einfachsten. Der Compiler wird durch Anwählen des Menüpunktes 'Compile...' im Oberon-

Menü gestartet. Genaueres hierzu finden Sie in den zwei vorhergehenden Kapiteln 3 und 4.

Aufruf von der Workbench

Um den Compiler von der Workbench zu starten muß zunächst das Piktogramm des Quelltexts, der übersetzt werden soll, angewählt werden. Sollen gleich mehrere Quelltexte übersetzt werden, können nun bei gedrückter Umschalttaste (Shift) weitere Piktogramme von Quelltexten angewählt werden. Der Compiler wird nun durch einen Doppelklick auf sein Piktogramm (die Windmühle) bei gedrückter Umschalttaste gestartet.

Es ist meist nicht sinnvoll, den Compiler von der Workbench aus durch einen Doppelklick zu starten ohne vorher einen Quelltext angewählt zu haben. Wird der Compiler so gestartet öffnet er sein Textfenster und wartet auf eine Eingabe wie bei einem Aufruf von der Shell ohne Argument (siehe unten).

Über Merkmale (Tool Types) des Piktogramms des Compilers kann man verschiedene Optionen setzen oder löschen. Gesetzt werden sie, indem sie auf 'TRUE' gesetzt werden. Gelöscht werden sie entsprechend, wenn sie auf 'FALSE' gesetzt werden. Die Optionen sind im Einzelnen:

Option:	Bedeutung:
STACKCHK	Stackkontrolle
OVFLCHK	Überlaufskontrolle
RANGECHK	Bereichskontrolle
CASECHK	Case-Index-Kontrolle
RETURNCHK	Return-Kontrolle
NILCHK	NIL-Zeiger-Kontrolle
ODDCHK	Ungerade-Zeiger-Kontrolle
TYPECHK	Typkontrolle
CLEARVARS	Lokale Variablen löschen
SMALLCODE	Kleines Codemodell
SMALLDATA	Kleines Datenmodell

```
MC68010             Code für MC68010 erzeugen
MC68020             Code für MC68020 erzeugen
MC68030             Code für MC68030 erzeugen
MC68881             Code für FPU MC68881/2 erzeugen
GARBAGECOLLECTOR    Garbage-Collector benutzen
EXTENSIONS          Spracherweiterungen benutzen
DEBUG               Code für Debugger ODebug
NEWSYMBOLS          Neue Symboldatei erzeugen
ICONS               Piktogramme erzeugen
```

Die ersten achtzehn Optionen (*STACKCHK* bis *DEBUG*) werden in Kapitel 14 genau beschrieben.

Die vorletzte Option, *NEWSYMBOLS*, erlaubt es dem Compiler, eine neue Symboldatei zu erzeugen. Wird es durch die seit der letzten Compilation am Quelltext gemachten Änderungen nötig, eine neue Symboldatei zu erzeugen, und ist diese Option nicht gesetzt, wird erst mit einer Dialogbox gefragt, ob eine neue Symboldatei erzeugt werden darf. Wird dies von Benutzer abgelehnt, erzeugt, der Compiler eine entsprechende Fehlermeldung.

Die letzte Option, *ICONS*, gibt an, ob der Compiler für die Dateien, die er erzeugt, Piktogramme erzeugen soll. Diese Option sollte beim Arbeiten mit der Workbench gewöhnlich gesetzt sein.

Aufruf von der Shell

Wer mit der Shell arbeitet, sollte das aktuelle Verzeichnis zunächst auf das Verzeichnis setzen, das die Quelltexte oder das 'txt'-Unterverzeichnis des bearbeiteten Projektes enthält.

Der Compiler benötigt gewöhnlich mehr als 4000 Bytes Stapelspeicher (Stack). Die Größe des Stapelspeichers sollte daher mit dem Shell-Befehl 'stack' auf einen Wert zwischen 20000 und 30000 gesetzt werden. Benutzen Sie den Compiler oft von der Shell aus, ist es wohl

sinnvoll, diesen Befehl in die Datei 'S:Shell-Startup' einzufügen, damit er nicht ständig neu eingegeben werden muß.

Nun kann der Compiler mit folgender Aufrufsyntax gestartet werden:

```
OBERON {   -{svbcrnotpzmd1238igeya}
         | ("SET"|"CLEAR") <Option>
         | <Quelltext>              }
```

Dabei ist *<Quelltext>* der Name des Quelltextes, der compiliert werden soll. Es können auch mehrere Quelltexte angegeben werden, die dann nacheinander compiliert werden. Die Endung '.mod' des Quelltextnamens kann weggelassen werden, sie wird automatisch von Compiler angehängt. Befindet sich der Quelltext im Unterverzeichnis 'txt', wird er automatisch von dort geholt, der Pfad muß nicht angegeben werden. So kann der Aufruf

```
Oberon txt/Test.mod
```

auch einfach durch

```
Oberon Test
```

abgekürzt werden.

Die Optionen bestehen aus einem Minuszeichen '-' gefolgt von einem oder mehreren Buchstaben, die für die verschiedenen Optionen stehen. Dabei verändert die Angabe einer Option den Zustand dieser jeweils. War eine Option gelöscht, so wird sie gesetzt, war sie gesetzt, wird sie entsprechend gelöscht.

Folgende Tabelle listet die Optionen, ihre Bedeutung und ihren Voreingestellte Wert. Wird eine Option nicht angegeben, wird der voreingestellte Wert verwendet.

Option:	Bedeutung:	Vorgabe:
-s	Stackkontrolle	TRUE
-v	Überlaufskontrolle	TRUE
-b	Bereichskontrolle	TRUE
-c	Case-Index-Kontrolle	TRUE
-r	Return-Kontrolle	TRUE
-n	NIL-Zeiger-Kontrolle	TRUE
-o	Ungerade-Zeiger-Kontrolle	FALSE
-t	Typkontrolle	TRUE
-z	Lokale Variablen löschen	TRUE
-m	Kleines Codemodell	FALSE
-d	Kleines Datenmodell	FALSE
-1	Code für MC68010 erzeugen	FALSE
-2	Code für MC68020 erzeugen	FALSE
-3	Code für MC68030 erzeugen	FALSE
-8	Code für FPU MC68881/2 erzeugen	FALSE
-a	Garbage-Collector benutzen	TRUE
-e	Spracherweiterungen benutzen	TRUE
-g	Code für Debugger ODebug	FALSE
-y	Neue Symboldatei erzeugen	TRUE
-i	Piktogramme erzeugen	FALSE

Die ersten achtzehn Optionen ('-s' bis '-g') werden in Kapitel 14 genau beschrieben.

Die vorletzte Option, '-y', erlaubt es dem Compiler, eine neue Symboldatei zu erzeugen. Ist es durch die seit der letzten Compilation am Quelltext gemachten Änderungen nötig geworden, eine neue Symboldatei zu erzeugen, und ist diese Option nicht gesetzt, so fragt der Compiler mit einem Dialogfenster nach, ob dennoch eine neue Symboldatei erzeugt werden soll. Wird dies von Benutzer abgelehnt, erzeugt der Compiler eine entsprechende Fehlermeldung.

Die letzte Option, '-i', gibt an, ob der Compiler für die Dateien, die er erzeugt, Piktogramme erzeugen soll. Diese Option ist beim Arbeiten mit der Shell gewöhnlich nicht nötig.

Die angegebenen Optionen beziehen sich auf alle Quelltexte, die hinter ihnen angegeben werden. Beispiel:

```
Oberon -bs Test1 -sv Test2
```

Hier wird Test1 ohne, Test2 jedoch mit Stapelkontrolle ('-s') übersetzt. Umgekehrt wird Test1 mit und Test2 ohne Überlaufskontrolle ('-v') übersetzt. Beide Module werden ohne Bereichskontrolle ('-b') übersetzt.

Hinter 'SET' bzw. 'CLEAR' können Optionen für die bedingte Compilation angegeben werden. So wird beim Aufruf

```
Oberon SET English Test.mod
```

der Quelltext 'Test.mod' mit der gesetzten Option 'English' übersetzt. Die Anwendung der bedingten Compilation wird in Kapitel 14 beschrieben.

Wird der Compiler ohne Angabe eines Quelltextes gestartet, gibt er als Aufforderung zur Eingabe 'in>' aus und wartet darauf, daß ein Quelltextname angegeben wird. Es können auch hier statt eines Namens Optionen angegeben werden. Beendet wird der Compiler durch einfaches Drücken der Eingabetaste <Return>.

Werden Optionen oft oder gar immer angegeben, empfiehlt es sich, für den Compileraufruf in der Shell-Startup ein 'Alias' anzulegen, um den Tippaufwand zu verringern. Ein Beispiel ist die folgende *Alias*-Anweisung:

```
Alias Ob Oberon -dm []
```

Manchmal ist es sinnvoll, Compileroptionen über Environment-Variablen anzugeben. So könnte die Environment-Variable *OberonOpt* die Optionen enthalten, die der Compiler verwenden soll. Dies kann man auch durch ein einfaches alias erreichen:

```
Alias Oberon Oberon $OberonOpt []
```

Dies funktioniert allerdings erst ab AmigaOS 2.0.

Aufruf des Compilers mit Batchdatei

Manchmal ist es nötig, mehrere Quelltexte direkt hintereinander zu compilieren. Dies geschieht am einfachsten, indem man die Namen der betroffenen Quelltexte in eine Textdatei, die sog. Batchdatei, schreibt. Dabei muß jeder Quelltextname in einer Zeile stehen. Auch können hier Optionen, die mit einem Minuszeichen, 'SET' oder 'CLEAR' beginnen, enthalten sein.

In eine Batchdatei geschrieben sieht der Aufruf 'Oberon -bs Test1 -sv Test' von Seite 5/7 folgendermaßen aus:

```
-bs
Test1
-sv
Test2
```

Diese Datei kann z.B. mit dem Namen 'bat' gespeichert werden. Um nun anhand dieser Batchdatei den Compiler zu starten, wird er wie folgt aufgerufen:

```
Oberon <bat
```

Die Anwendung von Batchdateien ist besonders dann sinnvoll, wenn das Make-Utility OMake (Kapitel 9) nicht sinnvoll angewendet werden kann, z.B. wenn mehrere Module geschrieben werden, die nicht zu einem einzigen Programm zusammengebunden werden sollen, oder wenn man mehrere Programme schreibt, die gemeinsame Module benutzen und daher oft alle neu compiliert werden müssen.

Arbeitsweise des Compilers

Während der Compiler einen Quelltext übersetzt zeigt er an, welche Dateien er einliest und welche er erzeugt. Beim Einlesen einer Datei gibt er den Text ' - <Dateiname>' und beim Erzeugen einer Datei entsprechend ' + <Dateiname>' aus. Dabei steht *<Dateiname>* jeweils für den Namen der Datei.

Während der Compilation wird zunächst der Quelltext des zu übersetzenden Moduls eingelesen. Nun benötigt der Compiler alle Symboldateien der importierten Module und liest auch diese ein. Automatisch wird auch jeweils die Symboldatei des Laufzeitmoduls *OberonLib* eingelesen. Zusätzlich werden manchmal auch die Module *GarbageCollector*, *MathFFP*, *MathTrans*, *MathIEEEDoubBas* und *MathIEEEDoubTrans* benötigt.

Nun beginnt der Compiler mit dem Übersetzen des Textes. Dabei wird für jedes KByte (1024 Bytes) an erzeugtem Code ein Punkt ('.') ausgegeben, damit man den Übersetzungsvorgang verfolgen kann.

Nach der Übersetzung prüft der Compiler, ob für den eben übersetzten Text bereits eine Symboldatei existiert. Ist dies der Fall, wird sie eingelesen und geprüft, ob sich an den exportierten Bezeichnern etwas geändert hat. Ist dies der Fall oder existiert noch keine Symboldatei, wird nun eine neue gespeichert.

Zuletzt versucht der Compiler den erzeugten Code zu optimieren. Dabei durchläuft er den Code mehrmals und gibt jeweils 'optimiere...' und die Anzahl der durch die Optimierung gesparten Bytes aus. Kann der Code nicht mehr weiter optimiert werden, wird er in eine Objekdatei gespeichert.

Zum Schluß gibt der Compiler noch die Größe des Programmcodes ('CODE'), des Konstantenbereichs ('DATA') und des Variablenbereichs ('BSS') aus. Die Größe der Variablenbereichs hat dabei keinen

Einfluß auf die Länge der Objektdatei und des ausführbaren Programms.

Wurden während der Compilation Fehler gefunden, so werden diese in die Datei '<Modulname>.modE' gespeichert. Die Fehler können dann z.B. mit OEd oder OErr (Kapitel 3, 4 und 7) betrachtet werden.

Beispiel einer Compilation

Wird das Modul 'Demo.mod' ohne Angabe von Optionen compiliert, erzeugt der Compiler z.B. die Ausgabe:

```
Amiga Oberon Compiler V3.0d
--- (c) 1992 by Fridtjof Siebert.
 - Demo.mod
 - Graphics.sym
 - Exec.sym
 - Utility.sym
 - Hardware.sym
 - Intuition.sym
 - InputEvent.sym
 - Timer.sym
 - KeyMap.sym
 - Dos.sym
 - OberonLib.sym
.
 - sym/Demo.sym
optimiere ...  106
optimiere ...   14
optimiere ...    2
optimiere ...    0
 + obj/Demo.obj
 CODE:  1912 DATA:   240 BSS:    432
tschüß!
```

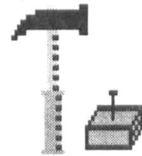
Verwendung des Compilers in einer Skriptdatei

Damit der Compiler in einer Skriptdatei eingesetzt werden kann, setzt er den Rückgabewert auf FAIL (20), wenn er einen der angegebenen Quelltexte nicht übersetzen konnte und eine Fehlerdatei erzeugt hat. Dies kann in einer Skriptdatei zum Compilieren ausgenutzt werden:

```
.KEY source
.BRA {
.KET }
FAILAT 21
OBERON {source}
IF FAIL
  OErr {source}
ELSE
  OLink {source}
ENDIF
```

Mit dieser Skriptdatei wird der Compiler mit dem angegebenen Quelltext als Argument gestartet. War die Compilation erfolgreich, wird das Programm gleich mit OLink gelinkt (Kapitel 6). Traten jedoch Fehler auf, werden diese mit OErr angezeigt (Kapitel 7).

6. Linken mit OLink



Die vom Compiler erzeugten Objektdateien sind Standard-Amiga-Objektdateien. Sie können daher mit dem Standardlinker 'BLink' zu einem ausführbaren Programm zusammengefügt werden.

Der Aufruf von 'BLink' ist jedoch nicht ganz einfach: 'BLink' kann nur von einer Amiga-Shell (CLI) aus gestartet werden. 'BLink' verlangt dabei eine komplette Liste der am Programm beteiligten Objektdateien. Diese ist nicht immer leicht zu erstellen und bringt meist viel Tipparbeit mit sich.

OLink vereinfacht das Linken im Vergleich zu 'BLink' stark. Es begnügt sich gewöhnlich mit dem Namen der Objektdatei des Hauptmoduls eines Programms und sucht sich alle anderen nötigen Objektdateien selbst.

Aufruf des Linkers

Aufruf aus dem Editor

Schreiben Sie Ihre Quelltexte mit dem Editor OEd, so ist der Aufruf von OLink aus dem Editor selbst gewöhnlich am einfachsten und bequemsten. Nach der Compilation des Hauptmoduls wird OLink durch Anwählen des Menüpunktes 'Link...' im Oberon-Menü gestartet. In den Kapiteln 3 und 4 wurde der Aufruf von OLink auf diese Weise genauer beschrieben.

Aufruf von der Workbench

Von der Workbench kann OLink sehr leicht gestartet werden indem das Piktogramm der Objektdatei des Hauptmoduls des Programms, das erzeugt werden soll, doppelklickt wird. Damit dies funktioniert,

muß sich OLink jedoch im logischen Verzeichnis 'OBERON:' befinden.

Aufruf von der Shell

Aufruf:

```
OLink {  [[FROM|MAIN] <Objektdatei>] [-mdsai]
         [OBJ|LIBRARY|LIB <Datei>]
         [DEFINE <Symbol>=<Symbol>|<Wert>]
         [WITH <Datei>] [TO <Datei>]
         [SMALLCODE|SD] [SMALLDATA|SD]
         [SMALL] [NOGC] [ICONS]                }
```

Dabei ist *<Objektdatei>* die Objektdatei des Hauptmoduls des Programms, das gelinkt werden soll.

Ähnlich wie beim Compiler können hier die Endungen '.obj', '.objs', '.obja' bzw. '.objsa' und der Pfad 'obj/' weggelassen werden. Der Aufruf

```
OLink obj/Test.obj
```

kann daher auch einfach als

```
OLink Test
```

abgekürzt werden.

Die synonymen Argumente *FROM* und *MAIN* sollten nur angegeben werden, wenn man Programme linken möchte, die nicht in Oberon geschrieben wurden. Sie verhindern die Suche nach der Objektdatei des Hauptmoduls in den in 'OBERON:Path' angegebenen Verzeichnissen.

Die Optionen, die aus einem Minuszeichen gefolgt von einem oder

mehreren Buchstaben bestehen, haben die im folgenden beschriebenen Bedeutungen:

Option: Wirkung:

-m Diese Option sollte angegeben werden, wenn das Programm mit dem kleinen Code-Modell (Kapitel 14) compiliert wurde. Sie bewirkt, daß alle Code-Hunks zu einem Hunk zusammengefügt werden.

Auch mit dem großen Code-Modell compilierte Programme können durch diese Option etwas verkürzt werden, es entstehen jedoch die gleichen Probleme wie bei '-s'.

-d Diese Option muß gesetzt werden, wenn mit dem kleinen Datenmodell (Compileroption '-d' oder Piktogramm-Merkmal 'SMALLDATA=TRUE', Kapitel 14) compiliert wurde. Wird diese Option angegeben, verwendet OLink die Objektdateien mit der Endung '.objs'. Zudem bestimmt OLink hier die Größe und den Typ des Speichers, den die globalen Variablen belegen.

Beim Linken mit dieser Option setzt OLink das 'pure'-Flag des erzeugten Programms und markiert es so als reentrante und residentfähige Programme.

-s Alle Code-, Daten- und BSS-Hunks (ein Hunk ist ein Block innerhalb einer Objektdatei, genaueres über den Aufbau von Objektdateien kann in [AmigaDos 91] nachgelesen werden) werden zu jeweils einem Hunk zusammengefügt, wenn diese Option gesetzt ist. Dadurch wird das ausführbare Programm etwas kürzer.

Dies hat jedoch den Nachteil, daß ein mit dieser Option gelinktes Programm nur noch aus großen Programmblöcken

besteht. Das Programm kann nur dann geladen und gestartet werden, wenn es große zusammenhängende Speicherbereiche gibt. So kann es bei fragmentiertem Speicher unmöglich sein, das Programm zu starten, obwohl noch genügend Speicher vorhanden wäre, dieser Speicher jedoch nur in kleineren Blöcken zur Verfügung steht.

-a Diese Option muß gesetzt werden, wenn das Programm ohne Verwendung des Garbage-Collectors übersetzt wurde (mit der Compileroption '-a' oder dem Piktogramm-Merkmal 'GARBAGECOLLECTOR=FALSE'). Dann verwendet OLink die Objektdateien mit der Endung '.obja' bzw. '.objsa'.

-i Ist diese Option gesetzt, erzeugt OLink ein Piktogramm für das ausführbare Programm.

Die Optionen können auch als Parameter ausgeschrieben werden. Dabei sind die Optionen gleichwertig mit den Parametern:

```
Option:  Ausgeschrieben:
-s       SMALL
-m       SMALLCODE oder SC
-d       SMALLDATA oder SD
-a       NOGC
-i       ICONS
```

Hinter den synonymen Parametern 'OBJ', 'LIBRARY' und 'LIB' können zusätzliche Objektdateien angegeben werden, die z.B. in Assembler geschriebene Routinen enthalten und von einem der Oberonmodule wie in Kapitel 15 beschrieben verwendet werden. OLink kann diese Objektdateien normalerweise nicht selbst finden, da in den Oberon-Objektdateien nur die Namen der importierten Oberonmodule enthalten sind.

Wird OLink ohne die Angabe einer Objektdatei gestartet, erwartet es, ähnlich wie der Compiler beim argumentlosen Start, die Eingabe des Namens einer Objektdatei.

Fortgeschrittene Parameter

Die in diesem Abschnitt beschriebenen Parameter sind für das gewöhnliche Arbeiten mit dem Oberon-Compiler und OLink nicht von Bedeutung. Sie werden unterstützt um eine bessere Kompatibilität zum Standardlinker 'BLink' herzustellen. Programmierer, die lediglich mit Amiga Oberon arbeiten möchten, können diesen Abschnitt ruhig überspringen.

DEFINE <Symbol>=<Symbol>|<Wert>

Hiermit können neue Symbole definiert werden. Ihnen kann entweder der Wert eines anderen Symbols zugewiesen werden, oder es kann ein neuer Wert als Dezimal- oder als Hexadezimalzahl (gefolgt von einem 'H') angegeben werden.

WITH <Datei>

Nach dem Parameter WITH wird eine Datei angegeben, die zusätzliche Parameter für OLink enthält. Die Syntax und Namen der Parameter in der WITH-Datei sind identisch mit denen beim direkten Aufruf von OLink. In der WITH-Datei dürfen jedoch zusätzliche Zeilenumbrüche vorkommen. Sie darf beliebig lang sein und auch weitere WITH-Anweisungen enthalten.

TO <Datei>

Optional kann hiermit der Name der Zieldatei angegeben werden. Wird er nicht angegeben, so wird beim Linken von Oberon-Programmen der Name der Objektdatei des Hauptmoduls ohne Endung verwendet. Wird kein Oberon-Programm ge-

linkt und wird mit 'TO' kein Name der Zieldatei angegeben, bekommt sie den Namen der Objektdatei mit der Endung '.bin'.

Arbeitsweise von OLink

Nachdem OLink gestartet wurde, durchsucht es die Objektdatei des Hauptmoduls nach Verweisen auf andere Objektdateien anderer Module. Diese Objektdateien werden dann jeweils geladen und ebenfalls untersucht, bis keine Verweise auf neue Objektdateien mehr gefunden werden.

Wurden zusätzliche Objektdateien (mit den Parametern 'OBJ', 'LIBRARY' oder 'LIB') angegeben, werden diese auch noch eingelesen.

Beim Linken von Programmen, die mit dem kleinen Daten-Modell compiliert wurden, bestimm OLink vor dem Erzeugen des ausführbaren Programms noch die Größe und der Typ des Speichers der globalen Variablen.

Nachdem alle Objektdateien gefunden wurden, wird das ausführbare Programm erzeugt und gespeichert. Dabei werden alle nicht verwendeten Hunks entfernt, so daß das Programm möglichst klein wird (optimierendes Linken). Existiert ein Unterverzeichnis 'bin' wird das ausführbare Programm dort abgelegt. Ansonsten kommt es in das aktuelle Verzeichnis.

Beispiel

Wurde der Quelltext 'Demo.mod' compiliert und wird das Programm nun mit 'OLink Demo' gelinkt, erzeugt OLink die Ausgabe:

```
Oberon Linker V3.0d
--- (c) 1992 by Fridtjof Siebert.
 - obj/Demo.obj
 - Oberon:obj/Dos.obj
 - Oberon:obj/Utility.obj
 - Oberon:obj/OberonLib.obj
 - Oberon:obj/Timer.obj
 - Oberon:obj/Intuition.obj
 - Oberon:obj/Graphics.obj
 + Demo
 CODE:   3144 DATA:    308 BSS:    728
 PROGRAM:    4308
tschüß!
```

Die Zahlen hinter 'CODE:', 'DATA:' und 'BSS:' geben die insgesamte Länge der Code, Daten und BSS-Bereiche des Programms an. Die Länge BSS-Bereiche trägt dabei nicht zur Programmlänge bei, sondern wird nur beim Starten des Programms benötigt.

Beim Linken mit dem kleinen Datenmodell wird zusätzlich hinter 'VAR:' die Größe des Variablenbereichs angegeben, der erst beim Programmstart angefordert wird.

Der Wert hinter 'PROGRAM:' ist die Dateilänge des Programms.

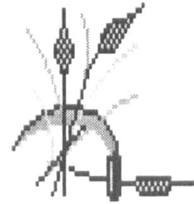
Verwendung von OLink in einer Skriptdatei

Damit Olink in einer Skriptdatei eingesetzt werden kann, setzt es den Rückgabewert auf FAIL (20), wenn es ein Programm nicht linken konnte. Der Grund hierfür kann z.B. sein, daß OLink eine nötige Objektdatei nicht finden konnte.

Dies kann in einer Skriptdatei zum Linken und Ausführen eines Programms ausgenutzt werden:

```
.KEY file
.BRA {
.KET }
FAILAT 21
OLINK {file}
IF FAIL
  ECHO "Konnte {file} nicht Linken"
ELSE
  ECHO "Starte {file}:"
  RUN {file}
ENDIF
```

7. Anzeigen von Fehlern mit OErr



Leider ist kaum jemand so perfekt und kann Programme schreiben, die vom Compiler ohne Probleme sofort übersetzt werden. Der Grund sind meist einfache Tippfehler, vergessene Deklarationen oder in schlimmeren Fällen sogar Denkfehler und Typinkompatibilitäten.

Der Compiler erzeugt in einem solchen Fall eine Fehlerdatei mit der Endung '.modE'. Wer den Editor OEd verwendet, kann die Fehler direkt im Editor betrachten. Wie dies genau geschieht wird im vierten Kapitel beschrieben. Wer jedoch einen anderen Texteditor OEd vorzieht, hat oft nicht die Möglichkeit, die fehlerhaften Textstellen und die Fehlerursachen so schnell zu finden. Hier hilft das Programm OErr.

Aufruf von OErr

Aufruf von der Shell

Aufruf:

```
OErr { <Quelltext> }
```

Als Argument erwartet OErr eine Liste von Quelltextnamen. Gibt man keinen Namen an, so wartet OErr, wie der Compiler, auf die Eingabe eines Namens.

Auch hier brauchen die Endung '.mod' und evtl. der Pfad 'txt/' nicht angegeben zu werden.

Aufruf von der Workbench

Von der Workbench muß zunächst das Piktogramm des fehlerhaften Textes angewählt, und danach das Piktogramm von OErr bei gedrückter Umschalttaste doppelgeklickt werden. OErr öffnet dann ein Fenster und gibt dort die Fehlerliste aus.

Beispiel

Nach dem Aufruf 'OErr Test' kann die Ausgabe folgendermaßen aussehen:

```
Oberon Fehlerlister V3.0d
--- (c) 1992 by Fridtjof Siebert.
 - OBERON:Fehler-Meldungen
in> test
 - txt/test.mod
 - txt/test.modE

  -------

  4:   WriteString("Hallo");
                  ^
      25: Bezeichner nicht definiert
     121: Bezeichner sollte Prozedur sein
                    ^
      27: ")" erwartet

  -------

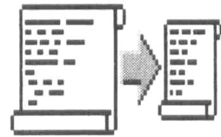
  5:   WriteLn;
             ^
      25: Bezeichner nicht definiert
     121: Bezeichner sollte Prozedur sein
in>

tschüß!
```

Die Zahlen links (hier '4:' und '5:') geben die Zeilenummern der fehlerhaften Textstellen an. Der Pfeil in den Zeilen darunter gibt die Position des Zeichens an, an dem der Fehler auftrat. Darunter werden jeweils die Fehlermeldungen ausgegeben, die an dieser Position aufgetreten sind. Sie beginnen mit der Nummer des aufgetretenen Fehlers (hier 25, 121 und 27) gefolgt von der Fehlermeldung im Klartext.

Ein Fehler kann Folgefehler verursachen, da die gemeinsame Ursache verschiedener Fehler vom Compiler nicht erkannt werden kann. So entstand hier in Zeile vier die Fehlermeldung "27: ')' erwartet", da der Compiler nicht wissen kann, das die eigentlich gemeinte Prozedur *io.WriteString* einen Parameter besitzt.

8. Das Hilfsprogramm ModToDef



Im Gegensatz zu Modula-2 kennt Oberon keine Definitionsmodule. Stattdessen ist jedes Modul eine einzige Textdatei, die die Funktionen der Definitions- und Implementationsmodule von Modula-2 vereint. Alle exportierten Bezeichner, die in Modula-2 im Definitionsmodul aufgelistet werden, werden hier schlicht mit einer Exportmarkierung ('*', bei Oberon-2 auch '-', genaueres dazu in den Kapiteln 12 und13 und in [Reiser 92]) versehen. Dadurch wird das Schreiben von Modulen leichter und übersichtlicher, man muß nicht mehr zwischen zwei Textdateien hin- und herwechseln, sondern arbeitet nur noch mit einem Text der dann auch den gesamten Quelltext eines Moduls darstellt.

Es gibt jedoch einen wichtigen Vorteil der Definitionsmodule, den man auch beim Arbeiten in Oberon nicht vermissen möchte: Definitionsmodule sind kompakte, übersichtliche Referenzen über die nach außen sichtbaren Bezeichner eines Moduls. Sie verbergen für die Anwendung eines Modules unnötige Informationen über die Art ihrer Implementierung.

ModToDef ist ein Programm, daß Definitionsdateien automatisch aus den Quelltexten von Oberon-Modulen erzeugt. Die Definitionsmodule müssen also nicht, wie in Modula-2, von Hand geschrieben werden, sondern werden automatisch erzeugt.

Eine Definitionsdatei in Oberon wird von keinem Programm benötigt und kann von keinem Programm bearbeitet werden. Ihre Aufgabe ist nur die Dokumentation von Modulen, sie werden daher nur von Programmierern und Anwendern von Modulen gelesen, die eine kurze Übersicht benötigen.

Aufruf von ModToDef

Aufruf von der Workbench

Von der Workbenchoberfläche aus wird ModToDef ähnlich wie der Compiler aufgerufen: Zunächst muß das Piktogramm des Quelltextes, dessen Definitionsmodul erzeugt werden soll, angewählt werden. Danach wird ModToDef durch einen Doppelklick auf sein Piktogramm bei gehaltener Umschalttaste (Shift) gestartet.

Durch erweiterte Anwahl bei gedrückter Umschalttaste können auch die Definitionsdateien mehrerer Quelltexte gleichzeitig mit ModToDef erzeugt werden.

Aufruf von der Shell

Aufruf:

```
ModToDef { <Quelltext> }
```

Es wird einfach ModToDef mit dem Namen der Quelltexte, aus denen Definitionsmodule erzeugt werden sollen, als Argumente gestartet. Werden mehrere Quelltextnamen übergeben, werden aus allen angegebenen Quelltexten Definitionsmodule erzeugt.

Die Endung '.mod' der Quelltexte kann auch hier weggelassen werden.

Wird ModToDef ohne Argument gestartet, so wartet es wie der Compiler auf die Eingabe des Namens eines Quelltextes.

Arbeitsweise von ModToDef

ModToDef liest zunächst den Quelltext ein. Dieser wird dann durchwandert, wobei sich ModToDef alle exportierten Bezeichner merkt.

Die typgebundene Prozeduren von Oberon-2 Modulen (siehe Kapitel 13, [Reiser 92] und [Mössenböck 91]) werden den Typbezeichnern ihrer Zieltypen zugeordnet und in die RECORD-Deklarationen dieser Typen eingetragen, so daß in dem Definitionsmodul der Zusammenhang zwischen den typgebundenen Prozeduren und den RECORDs besonders deutlich wird.

Zuletzt speichert ModToDef die Definitionsdatei noch unter dem Namen des Quelltextes, jedoch mit der Endung '.def'. Damit man ein Definitionsmodul leicht von einem gewöhnlichen Modul unterscheiden kann, beginnt es mit 'DEFINITION' statt 'MODULE'.

Damit ModToDef die Importliste des Definitionsmoduls korrekt erzeugen kann, müssen alle importierten Module, von denen Typen in der Definition von exportierten Bezeichnern abhängen, mit einem '*' markiert werden.

Beispiel:

```
MODULE Test;
IMPORT
  I * := Intuition;
VAR
  w * : I.Window;
BEGIN
  ...
END Test.
```

Beispiel

Wird aus dem der Quelltext des Moduls Lists (siehe Kapitel 19) mit 'ModToDef Lists' ein Definitionsmodul erzeugt, ist dies Ausgabe von ModToDef:

```
Module To Definition Converter V3.0d
--- (c) 1992 by Fridtjof Siebert.
in> Lists
 - Lists.mod
 + Lists.def
in>

tschüß!
```

Das erzeugte Definitionsmodul ist

```
DEFINITION Lists;

IMPORT
  BT := BasicTypes;

TYPE
  NodePtr = POINTER TO Node;
  Node = RECORD (BT.ANYRec)
    next : NodePtr;
    prev : NodePtr;
  END;
  ListPtr = POINTER TO List;
  List = RECORD (BT.COLLECTIONRec)
    head : NodePtr;
    tail : NodePtr;
    PROCEDURE (list:ListPtr) Add(x: BT.ANY);
    PROCEDURE (list:ListPtr) Remove(x: BT.ANY);
    PROCEDURE (list:ListPtr) nbElements(): LONGINT;
    PROCEDURE (list:ListPtr) isEmpty(): BOOLEAN;
    PROCEDURE (list:ListPtr) Do(p: BT.DoProc;
                                par: BT.ANY);
```

```
  END;
  DoProc = PROCEDURE(n: NodePtr);

PROCEDURE Init(VAR list: List);
PROCEDURE AddHead(VAR list: List; n: NodePtr);
PROCEDURE AddTail(VAR list: List; n: NodePtr);
PROCEDURE Remove(VAR list: List; n: NodePtr);
PROCEDURE RemHead(VAR list: List): NodePtr;
PROCEDURE RemTail(VAR list: List): NodePtr;
PROCEDURE AddBefore(VAR list: List;
                    n, x: NodePtr);
PROCEDURE AddBehind(VAR list: List;
                    n, x: NodePtr);
PROCEDURE Empty(VAR list: List): BOOLEAN;
PROCEDURE CountElements(VAR list: List): LONGINT;
PROCEDURE DoForward(VAR list: List; proc: DoProc);
PROCEDURE DoBackward(VAR list: List;
                     proc: DoProc);
PROCEDURE Next(VAR n: NodePtr): BOOLEAN;
PROCEDURE Previous(VAR n: NodePtr): BOOLEAN;
PROCEDURE Succ(VAR n: NodePtr);
PROCEDURE Pred(VAR n: NodePtr);
PROCEDURE Head(VAR list: List): NodePtr;
PROCEDURE Tail(VAR list: List): NodePtr;

END Lists.
```

9. Das Make-Utility OMake



Ein Make-Utility ist ein Programm, das das Compilieren von Modulen bei großen Programmprojekten stark vereinfacht. Wurden Module eines Projektes verändert, kann das Make-Utility dazu verwendet werden, alle Module, die bedingt durch diese Änderungen neu compiliert werden müssen, zu finden und automatisch zu übersetzen und zu einem lauffähigen Programm zusammenzufügen.

Aufruf von OMake

Aufruf aus dem Editor

Schreiben Sie Ihre Programme mit dem Editor OEd, so ist der Aufruf von OMake aus dem Editor selbst gewöhnlich am einfachsten und bequemsten. Bevor OMake aufgerufen werden kann, muß der Quelltext gespeichert werden. Nun kann dann Make-Utility mit 'Make...' des Oberon-Menüs gestartet werden. Genaueres zum Aufruf von OMake aus dem Editor steht in den Kapiteln 3 und 4.

Aufruf von der Workbench

Von der Workbench kann OMake durch erweitertes Anwählen (zusammen mit der Umschalttaste) des Piktogramms des Quelltextes des Hauptmoduls und des Piktogramms von OMake (kleine Windmühlen) gestartet werden.

Die Compileroptionen, die OMake beim Aufruf des Compilers verwenden soll, können über Merkmale (Tool Types) im Piktogramm von OMake eingestellt werden. Die Merkmale haben die gleichen Bezeichnungen wie die des Compilers (die Merkmale des Compiler-Piktogramms wurden in Kapitel 5 beschrieben).

Aufruf von der Shell

Beim Start von OMake aus einer Shell heraus können zusätzlich eine Reihe von Optionen angegeben werden.

Aufruf:

```
OMake { [c-{svbcrnotpzmd1238igeya}]
        [l-smdai] <Quelltext> [ALL] [DONTLINK]
        [OBJ <Objektdatei>] [(SET|CLEAR) Option] }
```

Dabei gibt *<Quelltext>* den Namen des Hauptmoduls an, das 'gemacht' werden soll. Bis auf *OBJ*, *SET* und *CLEAR* dürfen alle Argumente maximal einmal vorkommen.

Die Argumente haben folgende Bedeutungen:

c-[svbcrnotpzmd1238igeya]

Die Buchstaben nach dem Minuszeichen geben die Compileroptionen an, die beim Aufruf des Compilers verwendet werden sollen.

l-[smdai]

Die Buchstaben nach dem Minuszeichen geben die Linkeroptionen an, die beim Aufruf von OLink verwendet werden sollen.

ALL

Diese Option gibt an, daß auf jeden Fall alle an dem Programm beteiligten Module neu compiliert werden sollen. *ALL* sollte man z.B. dann angeben, wenn man ein Programm für die FPU (MC68881/2) völlig neu compilieren möchte. Gewöhnlich würde OMake lediglich die veränderten Quelltexte neu übersetzen.

DONTLINK

Soll OMake am Schluß des Make-Vorgangs OLink nicht aufrufen, muß diese Option gesetzt werden. Dies ist z.B. dann sinnvoll, wenn man das Make nur auf ein Untermodul anwendet und noch kein Interesse am fertigen Programm hat.

OBJ <Objektdatei>

Dieses Argument wird direkt an OLink weitergeleitet. Es dient zur Angabe zusätzlicher Objektdateien, die etwa Assemblerroutinen enthalten. Genaueres in Kapitel 6 über OLink und Kapitel 16 über die Einbindung von externen Objektdateien.

(SET | CLEAR) <Option>

Die Option *<Option>* für bedingte Compilation wird beim Aufruf des Compilers gesetzt bzw. gelöscht. Siehe hierzu Kapitel 5 und Kapitel 14.

Arbeitsweise von OMake

OMake untersucht den Quelltext des Hauptmoduls und aller von diesem Modul direkt oder indirekt importierten Module. Die Quelltexte werden dabei im aktuellen Verzeichnis und in den in 'Oberon:Path' angegebenen Pfaden gesucht.

Alle Module, deren Quelltexte jünger sind als ihre Objektdateien, werden neu compiliert. Das gleiche geschieht mit Modulen, die Module importieren, deren Symboldateien jünger sind als die eigene Objektdatei.

Zum Schluß wird noch geprüft, ob das gerade 'gemachte' Programm bereits existiert und jünger ist als die älteste Objektdatei, aus der es

besteht. Ist dies der Fall, oder existiert das Programm noch nicht, wird OLink so aufgerufen, daß es gelinkt wird.

Da OMake den Compiler sehr oft aufruft, ist es sinnvoll, während dem Arbeiten mit OMake den Compiler mit dem Shell-Befehl 'resident' im Speicher zu halten. Ansonsten wird der Compiler evtl. sehr häufig nachgeladen. Auch ist hier die Anwendung des Resident-Managers (siehe Kapitel 10) meist sehr sinnvoll, damit nicht ständig dieselben Symboldateien nachgeladen werden.

Beispiele

Der Aufruf

```
OMake Test
```

compiliert alle Module, die Test.mod direkt oder indirekt importiert und die seit der letzten Compilation verändert wurden. Danach wird evtl. OLink aufgerufen und Test neu gelinkt. Dagegen compiliert der Aufruf

```
OMake c-md28 l-md Test ALL
```

alle Module, die Test.mod direkt oder indirekt importiert neu und linkt Test. Das erzeugte Programm ist eine für die Prozessorkombination MC68020/68881 optimierte Version von Test.

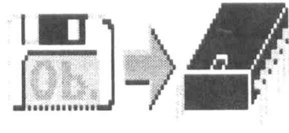
Probleme

Damit OMake korrekt arbeiten kann, dürfen die Erstellungsdaten der Quelltexte, der Objekt- und der Symboldateien nicht verändert werden. Dies geschieht z.B. beim Kopieren dieser Dateien mit dem Shell-Befehl 'copy', wenn man die Option 'clone' nicht angiebt. Besonders bei der Installation des Oberon-Compilers auf die Harddisk

muß man hier vorsichtig gewesen sein, sonst kann OMake fälschlicherweise auf die Idee kommen, die mitgelieferten Module neu zu compilieren.

Implementationslose Module (wie z.B. Exec.mod), die jünger sind als ihre Symboldatei, beim Compilieren jedoch keine neue Symboldatei erzeugen, werden immer neu compiliert. Abhilfe kann man hier nur schaffen, indem man das Datum der Symboldatei verändert, also z.B. indem man sie mit COPY in die RAM-Disk und wieder zurück an ihren ursprünglichen Ort kopiert.

10. Der ResidentManager



Ein großer Teil der zur Compilation benötigten Zeit wird für das Laden von Symboldateien aufgewendet. Besonders bei einer langsamen Harddisk oder gar bei der Verwendung von Diskettenlaufwerken steht die für das Laden der Symboldateien benötigte Zeit oft in keinem Verhältnis zur eigentlichen Compilationszeit.

Für das Laden der Symboldateien ist die 'oberonsupport.library' verantwortlich, die Sie bei der Installation des Compilers in das Verzeichnis 'LIBS:' kopieren mußten. Diese Library ist in der Lage, eine einmal geladene Symboldatei gleichzeitig mehreren Programmen zur Verfügung zu stellen, also etwa dem Compiler und dem Debugger ODebug gleichzeitig. Zudem kann sie Symboldateien im Speicher resident halten, wenn man dies von ihr verlangt.

Mit dem Programm ResidentManager kann der Library mitgeteilt werden, ob und welche Symboldateien resident gehalten werden sollen. ResidentManager benötigt dazu eine Liste der Namen der Module, deren Symboldateien resident gehalten werden sollen. Diese Liste wird als gewöhnliche Textdatei in der Datei 'Oberon:ResidentModules' gespeichert. Sie kann beispielsweise so aussehen:

```
OberonLib
Intuition
io
Display
Concurrency
```

Es werden die Symboldateien der aufgelisteten Module und die Symboldateien aller Module, von denen diese abhängen (die von diesen importiert werden) resident gehalten.

Hat man einen großen Arbeitsspeicher, ist es sinnvoll, schlicht alle Symboldateien resident zu halten. Damit hier die Datei 'Oberon:Resi-

dentModules' nicht unübersichtlich lang wird und nicht ständig angepaßt werden muß, kann sie hier einfach mit

```
all
```

abgekürzt werden.

Aufruf des ResidentManagers

Aufruf von der Workbench

Von der Workbench wird ResidentManager durch einen Doppelklick auf sein Piktogramm gestartet. Um den Speicher der residenten Symboldateien wieder freizugeben, kann ResidentManager einfach noch einmal gestartet werden.

Aufruf von der Shell

Aufruf:

```
ResidentManager [RESET]
```

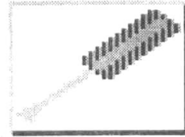
Der Aufruf von ResidentManager ohne Argument aktiviert die Liste der residenten Symboldateien. Nochmaliger Aufruf deaktiviert diese Liste wieder und gibt den Speicher der residenten Symboldateien wieder frei.

Oft möchte man den Speicher der residenten Symboldateien freigeben, um für eine Weile mehr freien Arbeitsspeicher zu haben, möchte die Liste der residenten Symboldateien jedoch nicht deaktivieren. Dazu wird ResidentManager mit dem Argument *RESET* aufgerufen. Auf diese Weise kann auch eine veränderte Liste 'Oberon:ResidentModules' aktiviert werden, wenn ResidentManager bereits gestartet war. Ein Aufruf von *ResidentManager RESET* ist

auch dann erforderlich, wenn 'von Hand' Symboldateien gelöscht oder kopiert wurden, damit die im Speicher gehaltenen Dateien immer mit den gespeicherten übereinstimmen.

Es empfiehlt sich gewöhnlich, den Aufruf des ResidentManagers in die Startup-Sequence aufzunehmen. Da der ResidentManager nur der 'oberonsupport.library' mitteilt, was sie zu tun hat, und selbst nicht weiterläuft, kann er in der Startup-Sequence ohne dem Shell-Befehl 'run' gestartet werden.

11. Der Library-Linker LibLink



Mit dem Programm LibLink ist es möglich, in Oberon geschriebene Module zu Amiga-Libraries und Amiga-Devices, so wie sie in den Verzeichnissen 'LIBS:' und 'DEVS:' zu finden sind, zu erstellen. Da diese Anwendung zweifellos für fortgeschrittene Amiga-Programmierer interesant ist, können Anfänger dieses Kapitel überspringen.

Wer LibLink benutzen möchte sollte ausreichend Vorwissen über den Aufbau von Libraries und Devices besitzen. Beschreibungen dazu sind in [RKM: Libraries 92] und [RKM: Devices 91] enthalten.

Installation

LibLink und alle für LibLink nötigen Dateien befinden sich im Unterverzeichnis 'LibLink' auf einer der Oberon-Disketten. Damit LibLink korrekt funktionieren kann, müssen sich die Objektdateien Library-Head.obj[s][a] und LibOberonLib.obj[s][a] in einem in 'Oberon:Path' angegebenen Verzeichnis befinden. Um dies zu erreichen kann z.B. der Pfad des Verzeichnisses 'LibLink' in 'Oberon:Path' aufgenommen werden.

Aufruf von LibLink

LibLink kann nur von der Shell aus sinnvoll verwendet werden, da es eine große Zahl an Argumenten benötigt.

Aufruf:

```
LibLink  { [[FROM|MAIN] <Objektdatei>] [-smdai]
           [OBJ|LIBRARY|LIB <Datei>]
           [DEFINE <Symbol>=<Symbol>|<Wert>]
           [WITH <Datei>] [TO <Datei>]
```

```
            [SMALLCODE|SC] [SMALLDATA|SD]
            [SMALL] [NOGC] [ICONS] [DEVICE]
            [PROC <Module.Name>]
            [OPEN <Module.Name>]
            [CLOSE <Module.Name>]
            [VERSION <n>] [REVISION <n>]
            [PRI <n>] [SIZE <n>]
            [XREF <Datei>]
            [IDSTRING <IDString>]               }
```

Die Optionen *-smdai* und die Argumente *FROM*, *MAIN*, *OBJ*, *LIBRARY*, *LIB*, *DEFINE*, *SMALLCODE*, *SC*, *SMALLDATA*, *SD*, *SMALL*, *NOGC* und *ICONS* haben dieselbe Bedeutung wie bei dem Linker OLink. Sie wurden in Kapitel 6 beschrieben.

<Objektdatei> ist hier die Objektdatei des Hauptmoduls, aus dem die Library bzw. das Device gebildet werden soll. Es muß alle Module, deren Prozeduren zu Library- oder Devicefunktionen werden, direkt oder indirekt importieren.

Die Bedeutung der anderen Argumente ist:

WITH <Datei>

Wie bei OLink kann hier eine Datei mit zusätzlichen Parametern angegeben werden. Diese Datei kann auch alle Parameter, die LibLink kennt, enthalten.

Die Anwendung von *WITH* ist bei LibLink in den meisten Fällen dringend nötig, da meist allein die Liste der Libraryfunktionen (Argument *PROC)* die Maximallänge der Shell-Eingabezeile übersteigt.

TO <Datei>

Der Name der Zieldatei muß beim Linken mit LibLink gewöhn-

lich angegeben werden, da er nicht immer im direkten Zusammenhang mit dem Hauptmodul steht. Wird keine Zieldatei angegeben, wird der Name des Hauptmodules mit der Endung '.library' bzw. '.device' verwendet.

DEVICE

Diese Option muß angegeben werden, wenn die Zieldatei ein Device werden soll. Ist sie nicht angegeben, wird eine Library gelinkt.

PROC <Modul.Name>

Mit *PROC* werden die Oberon-Prozeduren angegeben, die die Funktionen der Library bilden sollen. Da eine Library gewöhnlich aus mehreren Funktionen besteht, kann *PROC* mehrfach angegeben werden. Die zuerst angegebenen Prozeduren bekommen den betragsmäßig kleinsten Offsets relativ zur Basisadresse der Library bzw. des Devices. Die zuerst angegebene Prozedur hat den Offset -30, die weiteren -36, -42, etc.

Die Namen der Prozeduren müssen qualifiziert angegeben werden, d.h. sie bestehen aus dem Namen des Modul, in dem sie definiert wurden, einem Punkt und dem Prozedurnamen selbst. Die Prozeduren müssen im Oberon-Quelltext als exportiert gekennzeichnet sein.

Es können auch Prozeduren aus mehreren verschiedenen Modulen angegeben werden. Es müssen jedoch alle beteiligten Module direkt oder indirekt vom Hauptmodul importiert werden, oder selbst das Hauptmodul sein.

Wird ein Device gelinkt, so müssen als erste beiden Funktionen die Prozeduren für die Device-Funktionen *BeginIO* und *AbortIO* angegeben werden.

OPEN <Modul.Name>

Mit *OPEN* kann optional eine parameterlose Prozedur mit einem Ergebnis vom Typ *BOOLEAN* angegeben werden, die bei jedem Öffnen der Library oder des Devices aufgerufen wird. Das Ergebnis dieser Prozedur muß *FALSE* sein, wenn das Öffnen nicht möglich war, weil z.B. nicht genügend Speicher vorhanden war. Ansonsten muß es *TRUE* sein.

CLOSE <Modul.Name>

Hiermit kann eine optionale Prozedur angegeben werden, die bei jedem Schließen der Library bzw. des Devices aufgerufen werden soll.

VERSION <n>

Die Versionsnummer der Library bzw. des Devices wird auf *<n>* gesetzt. Wird *VERSION* nicht angegeben, wird als Versionsnummer 0 verwendet.

REVISION <n>

Die Revisionsnummer der Library bzw. des Devices wird auf *<n>* gesetzt. Wird *REVISION* nicht angegeben, wird geprüft, ob die bei *TO* angegebene Zieldatei bereits existiert und wenn ja, wird deren Revisionsnummer um eins erhöht und für die neue Zieldatei verwendet. Sonst wird als Revisionsnummer 0 verwendet.

PRI <n>

Die Priorität der Library bzw. des Devices wird auf den Wert *<n>* gesetzt.

SIZE <n>

Hier kann die Größe der Library- bzw. Device-Basisstruktur angegeben werden. Diese ist gewöhnlich SIZE(Exec.Library) = 34. Sollen zusätzliche Informationen in der Basisstruktur gespeichert werden, muß hier die Größe der verlängerten Struktur angegeben werden. Die Größe einer erweiterten Basisstruktur kann beispielsweise mit *io.WriteInt(SIZE(MyLibBase));* ermittelt werden.

XREF <Datei>

Es wird der Name einer Querverweis-Datei angegeben, die LibLink erzeugen soll. Diese Datei enthält dann Informationen über die Library und die Offsets ihrer Funktionen. Sie ist besonders Hilfreich beim Schreiben eines Interfacemoduls zur Library (siehe Kapitel 14). Beispiel einer Querverweisdatei:

```
(*
 * Oberon Library Linker Cross Reference Listing:
 *
 * Library:  intmathe.library
 * Version:        2
 * Revision:      12
 *)
MODULE Mathe;
IMPORT e := Exec;
VAR base*: e.LibraryPtr;

PROCEDURE Mathe.GGT      {base,-   30}();
PROCEDURE Mathe.KGV      {base,-   36}();
PROCEDURE Mathe.Sqrt     {base,-   42}();
PROCEDURE Mathe.Fak      {base,-   48}();
PROCEDURE Mathe.BinKoeff{base,-   54}();
PROCEDURE Mathe.Pow      {base,-   60}();

END Mathe.
```

IDSTRING <IDString>

Es kann der Identifikationstext *<IDString>* der Library bzw. des Devices angegeben werden. Dieser Text sollte aus dem Namen, der Versions- und Revisionsnummer und dem Datum der letzten Änderung bestehen. Ein korrektes Beispiel ist "MeineLibrary 3.23 (18-Jul-92)".

Wichtige Hinweise

Die Module, und alle von ihnen importierten Module, die zu Libraries oder Devices gebunden werden sollen, müssen folgende Eigenschaften haben:

- alle Prozeduren müssen wiedereintrittsfähig (reentrant) sein. Das heißt, sie dürfen globale Variablen nur bei besonderer Vorsicht verwenden, da die gleiche oder eine andere gleichzeitig aufgerufene Prozedur auch mit den globalen Variablen arbeiten kann. Eventuell müssen die Variablen mit einer Semaphore (siehe unter Exec in [RKM: Libraries 92]) geschützt werden.

- Die Prozeduren, die als Library- oder Devicefunktionen dienen sollen, dürfen keine Prozessorregister verändern. Dies geschieht durch Angabe der Option *(* SaveRegs+ *)* zu Beginn jeder Prozedur (Kapitel 14).

- Der Garbage-Collector (Kapitel 16) darf nicht verwendet werden, es sollte daher mit der Option '-a' oder mit dem Piktogramm-Merkmal *GARBAGECOLLECTOR=FALSE* compiliert werden.

- Das kleine Datenmodell darf nur mit großer Vorsicht verwendet werden. Beim kleinen Datenmodell muß zu Beginn jeder Prozedur, die als Library- oder Devicefunktion verwendet wird, der Zeiger auf die globalen Variablen mit OberonLib.SetA5 wiederhergestellt werden.

- Folgende Module dürfen nicht verwendet werden, sie sind nicht auf die Verwenung innerhalb einer Library ausgelegt:

Arguments	**Beep**	**Break**
BreakRq	**Concurrency**	**Debug**
Display	**FileReq**	**FileSystem**
GarbageCollector	**In**	**io**
LongRealInOut	**Mouse**	**NoGuru**
NoGuruRq	**OberonLib**	**Out**
RealInOut	**SecureDos**	

  Da alle Module automatisch OberonLib importierten, wird OberonLib von LibLink durch LibOberonLib ersetzt.

- Die Module sollten ohne Stackkontrolle übersetzt werden. Es ist jedoch nicht schlimm, wenn die Stackkontrolle bei manchen Modulen dennoch eingeschaltet ist, es wird dann lediglich Speicher und Rechenzeit verschwendet.

- Die Module dürfen nur während der Ausführung der BEGIN-Anweisungen mit HALT() abbrechen, überall sonst führt HALT() zu einem Absturz.

- Die BEGIN- und CLOSE-Anweisungen werden bei abgeschalteten Multitasking (Exec.Forbid) ausgeführt. Sie dürfen das Multitasking auf keinen Fall ermöglichen (indem Sie eine z.B. Exec.Wait oder eine Routine aus Dos verwenden).

- Laufzeitfehler führen zu einem Systemabsturz mit einer Alert-Meldung. Dies sollte jedoch kein Grund sein, den Überprüfungscode abzustellen, sondern vielmehr sollten die Module mit mehr Sorgfalt geschrieben und gut ausgetestet werden. Ein Fehler, der gleich zu einem Absturz führt ist immer besser als einer, den man erst daran erkennt, daß sich ein Programm in einer wichtigen Situation falsch verhält.

- Einige Variablen aus *OberonLib* enthalten in einer Library keine

sinnvollen Werte und dürfen nicht benutzt werden. Genaueres hierzu ist in Kapitel 26 zu finden.

Libraries können auf ein paar globale Variablen zugreifen, die im Assemblermodul LibraryHead definiert sind (der Zugriff auf Variablen von Assemblermodulen wird in Kapitel 15 beschrieben):

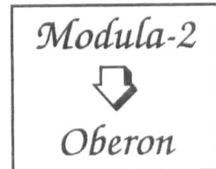
```
VAR
  libBase["LibraryHead.LibBase"]: Exec.LibraryPtr;
  segList["LibraryHead.SegList"]: Exec.BPTR;
  globals["LibraryHead.Globals"]: Exec.APTR;
```

libBase zeigt auf die Library-Struktur dieser Library (bei einem Device entsprechend auf die Device-Struktur). Wurde beim Linken das Argument *SIZE* angegeben, zeigt libBase auf eine erweiterte Struktur mit zusätzlichen Elementen. Dann sollte *libBase* als Zeiger auf eine Erweiterte Library- bzw. Device-Struktur definiert werden.

segList enthält einen Zeiger auf die Dos-Segmentliste der geladenen Library bzw. des Devices. Segmentlisten werden in [AmigaDOS 91] beschrieben.

Die Variable *globals* enthält den Zeiger auf den Bereich der globalen Variablen. *globals* ist vor allem beim kleinen Datenmodell wichtig, damit auf die globalen Variablen zugegriffen werden kann. OberonLib.SetA5 benutzt dafür *globals*.

12. Unterschiede zwischen Modula-2 und Oberon



Oberon ist eine Weiterentwicklung der Sprache Modula-2 [Wirth 85] und führt somit die lange Sprachentwicklung fort, die von Algol über Pascal zu Modula führte. Bei der Entwicklung von Oberon wurde besonderer Wert darauf gelegt, die Sprache zu vereinfachen und dabei gleichzeitig ihre Möglichkeiten zu erweitern. Dazu wurde sie um ein paar wenige, leistungsstarke Fähigkeiten ergänzt, während viele andere Fähigkeiten, die entweder überflüssig oder gar hinderlich geworden sind, gestrichen wurden. Das Ergebnis ist eine leicht erlernbare, einfache Sprache, die dennoch viele neue Möglichkeiten bietet und vielen bisherigen Sprachen überlegen ist.

Eine ausführliche Beschreibung der Sprache Oberon ist in [Reiser 92] enthalten. Eine kompakte Sprachdefinition ist in [Wirth 88] und in einer aktuelleren Fassung in [Wirth 90] enthalten.

Neue Fähigkeiten von Oberon

Die wohl wichtigste Neuerung in Oberon ist die Erweiterbarkeit von RECORD-Typen. In Modula war es bereits möglich, auf der Basis von bestehenden Prozeduren neue Prozeduren zu schreiben. Hierfür fehlte jedoch die Entsprechung bei den Typen. In Oberon ist es nun möglich, aufbauend auf bereits existierenden RECORD-Typen, den sogenannten Basistypen, neue RECORD-Typen zu definieren.

Die neu definierten Typen bleiben weiterhin kompatibel zu ihren Basistypen, die Prozeduren die für die alten Typen geschrieben wurden werden an die neuen 'vererbt'. Dieses Konzept macht Oberon zu einer objektorientierten Sprache.

Ein schönes Beispiel für die Anwendungsmöglichkeiten, die einem durch die Möglichkeit der Recorderweiterung geboten werden, sind

die Typen der Knoten in einem Baum. Zunächst definieren wir die nicht erweiterten Basistypen der Knoten:

```
TYPE
  Node = POINTER TO NodeDesc;
  NodeDesc = RECORD
    left, right: NodePtr;
    key: INTEGER;
  END;
```

Der Baum, der von diesen Knoten aufgebaut wird, soll aufsteigend nach dem Schlüssel *Node.key* sortiert werden. Eine Prozedur, die nach einem Knoten mit einem bestimmten Schlüssel in einem Baum sucht, sieht folgendermaßen aus:

```
PROCEDURE Find(root: Node;
                  key: INTEGER): Node;
BEGIN
  WHILE (root # NIL) & (root.key # key) DO
    IF root.key<key THEN
      root := root.right;
    ELSE
      root := root.left;
    END;
  END;
  RETURN root;
END Find;
```

Entsprechend sieht eine Prozedur zum Einfügen eines Elementes in den Baum aus:

```
PROCEDURE Insert(VAR root: NodePtr; node: Node);
BEGIN
  IF root=NIL THEN
    root := node
  ELSIF root.key > node.key THEN
    Insert(root.left,node);
```

```
  ELSE
    Insert(root.right,node);
  END;
END Insert;
```

Bisher unterscheiden sich die Programmstücke nicht entscheidend von Modula-2 Quelltexten. Sie machen jedoch bisher auch noch nichts direkt sinnvolles, da in dem Baum noch keine Daten gespeichert werden.

Nehmen wir an, wir wollen ein objektorientiertes Grafikprogramm schreiben, daß den bereits implementierten Baum zur Speichern der verschiedenen Grafikobjekte verwendet. Dazu definieren wir die Grafikobjekte als Erweiterungen des Knotentyps. So definieren wir Typen für Rechtecke und Kreise:

```
TYPE
  Rect = POINTER TO RectDesc;
  RectDesc = RECORD (NodeDesc)
    x,y,b,h: INTEGER;
  END;

  Circle = POINTER TO CircleDesc;
  CircleDesc = RECORD (NodeDesc)
    mx,my: INTEGER;
    r: INTEGER;
  END;
```

Diese zwei RECORD-Typen enthalten, zusätzlich zu den hier definierten Elementen, die Elemente *left, right und key* ihres Basistyps *NodeDesc*. Die beiden RECORD-Typen *RectDesc* und *CircleDesc* sind zuweisungskompatibel zum Typ *NodeDesc*. Entsprechend gilt dies auch für die Zeigertypen *Rect* und *Circle*, die Zuweisungskompatibel zu *Node* sind. Daher können die Grafikobjekte auch mit der Prozedur *Insert* von oben in den Baum eingefügt werden, ohne daß *Insert* angepaßt werden muß. In denselben Baum können auch gleichzeitig

sowohl Rechteck- als auch Kreisobjekte eingefügt werden. Das Ergebnis ist ein sogenannter inhomogener Baum.

Eine Prozedur zum Erzeugen eines neuen Rechtecks kann zum Beispiel folgendermaßen aussehen:

```
PROCEDURE NewRect(VAR root: Node; key: INTEGER);
VAR
  r: Rect;
BEGIN
  NEW(r);
  r.key := key;
  WaitForMouseClick;
  r.x := MouseX();        r.y := MouseY();
  WaitForMouseClick;
  r.b := MouseX() - r.x; r.h := MouseY() - r.y;
  Add(root,r);
END NewRect;
```

Wenn nun ein Grafikobjekt mit einem bestimmten Schlüssel gezeichnet werden soll, kann es mit der Prozedur *Find* gesucht werden. Nun muß jedoch noch festgestellt werden, welche Art von Grafikobjekt es ist, in diesem Fall also, ob es ein *Rect* oder ein *Circle* ist. Dies kann in Oberon mit einem Typ-Test mit dem Operator *IS* festgestellt werden. Dabei liefert der Ausdruck *v IS T* den *BOOLEAN*-Wert *TRUE*, wenn die Zeigervariable *v* auf ein Objekt vom Typ *T* zeigt. *node IS Rect* liefert also *TRUE*, wenn der Knoten ein Rechteckt ist.

Nachdem der Typ bestimmt wurde, möchte man auf die zusätzlichen Elemente in der Erweiterung zugreifen können. Dazu muß dem Compiler noch mitgeteilt werden, daß man die Zeigervariable nun so verwendet, als würde sie auf ein Objekt des erweiterten RECORD-Typs zeigen. Dies geschieht mit einem Typeguard, der in Runde Klammern hinter die Variable geschrieben wird: *v(T)*. In unserem Beispiel wird auf die Elemente der Rechtecks z.B. mit *node(Rect).x* zugegriffen.

Soll über ein größeres Programmstück hinweg ein Typeguard verwendet werden, kann man eine *WITH*-Anweisung verwenden, anstatt bei jedem Vorkommen der Variablen den Typeguard anzugeben.

Die Prozedur zum Zeichnen eines Grafikobjektes sieht nun also in etwa so aus:

```
PROCEDURE Draw(root: Node; key: INTEGER);
VAR
  node: Node;
BEGIN
  node := Find(root,key);
  IF node # NIL THEN
    IF node IS Rect THEN
      WITH node : Rect DO
        DrawRect(node.x,node.y,node.b,node.h);
      END;
    ELSIF node IS Circle THEN
      DrawCircle(node(Circle).mx,
                 node(Circle).my,
                 node(Circle).r);
    END;
  END;
END Draw;
```

Bei der Definition von exportierten RECORD-Typen müssen in Oberon alle Elemente, die nach außen sichtbar sein sollen, mit einem Sternchen gekennzeichnet werden. Auf diese hat man eine viel flexiblere Möglichkeit, Typen mit nach außen nicht sichtbarem oder nur teilweise sichtbarem Inhalt zu definieren. Modula-2 kannte nur opake Typen, deren Inhalt nach außen immer vollständig versteckt war.

Ein opaker Typ wird in Oberon einfach dadurch erzeugt, daß man einen Recordtypbezeichner exportiert, jedoch keines der Recordelemente markiert. Nach außen ist also nur ein leeres Record zu sehen.

Eine große Flexibilität in Oberon ergibt sich aus dem Prinzip des Typeinschlusses. Danach sind alle numerischen Typen in einer Hierarchie angeordnet. Die in der Hierarchie oben liegenden Typen 'enthalten' die weiter unten liegenden. Ihnen können somit die Werte der tiefer liegenden Typen zugewiesen werden. Beim Rechnen mit numerischen Werten werden die Operanden in den jeweils größten Typ umgewandelt, der die Typen beider Operanden enthält.

```
SHORTINT ⊆ INTEGER ⊆ LONGINT ⊆ REAL ⊆ LONGREAL
```

Nach der Deklaration der Variablen

```
VAR
  i: INTEGER;
  j: LONGINT;
  k: REAL;
  l: LONGREAL;
```

haben die folgenden Ausdrücke die angegebenen Typen:

```
Ausdruck:    Ergebnistyp:
k*i          REAL
i*l+j        LONGREAL
j/i          REAL
j DIV i      LONGINT
```

Das Ergebnis einer Division mit '/' ist immer ein Real-Typ, auch wenn die Operanden nur von Integer-Typen sind.

Mit den Variablen von oben können folgende Zuweisungen ausgeführt werden:

```
j := i;
k := j;
l :=  k+10;
```

Zuweisungen wie 'i := j' oder 'j := k' sind jedoch nicht möglich.

Prozedurtypen werden in Oberon ähnlich definiert wie Prozeduren selbst auch: Es werden auch hier Bezeichner für die Prozedurparameter angegeben. DieBezeichner werden, außer zur Dokumentation jedoch nicht benötigt, der Compiler beachtet sie nicht weiter. Beispiel:

```
TYPE Proc = PROCEDURE(i,j: INTEGER);
```

Fähigkeiten, die gegenüber Modula-2 fehlen

Typen

Variante Records und opake Typen gibt es in Oberon nicht mehr. Sie können die neuen Möglichkeiten der Erweiterung von Records und dem einzelnen Export der Recordelemente ersetzt werden.

Aufzählungstypen gibt es in Oberon nicht mehr. Sie müssen durch Integer-Konstanten und Variablen ersetzt werden.

Oberon kennt keine Unterbereichstypen mehr. Es muß stattdessen der jeweilige Basistyp verwendet werden.

Durch das Fehlen der Aufzählungs- und Unterbereichstypen verliert die Definition neuer Mengentypen ihren Sinn. Stattdessen gibt es in Oberon nur noch einen Standard-Mengentyp *SET* (diese Oberon-Implementierung kennt zusätzlich die Mengentypen SHORTSET und LONGSET).

In Oberon gibt es die Typen *CARDINAL* und *LONGCARD* nicht mehr. Es gibt kaum Modula-Programme, die den vollen Bereich der Cardinal-Zahlen ausgenutzt haben. Stattdessen kann also auch *INTEGER* verwendet werden. Durch das Fehlen der *CARDINALs* in Ober-

on hat die Inkompatibilität zwischen *CARDINAL* und *INTEGER*, die Modula-2 kennt, ein Ende.

Zeiger in Oberon dürfen nur auf Record- und Array-Typen zeigen (bei Amiga Oberon gibt es diese Einschränkung jedoch nciht).

Der Indextyp von Arrays ist in Oberon immer ein Integer-Typ. Bei der Definition von ARRAY-Typen werden nicht mehr zwei Grenzen, sondern wird nur noch die Länge des Feldes angegeben. Die Feldelemente reichen mit den Indizes von null bis zur Länge minus eins.

Module

Lokale Module gibt es in Oberon nicht mehr, da sie in Modula-2 praktisch nie eingesetzt wurden.

Bezeichner können in Oberon nicht mehr einzeln mit *FROM module IMPORT ident* importiert werden. Es muß stattdessen das gesamte Modul mit *IMPORT module* importiert und auf die Bezeichner qualifiziert mit *module.ident* zugegriffen werden. Um hierbei unnötig viel Tipparbeit zu vermeiden, kann der Modulname in der Importliste abgekürzt werden: *IMPORT m := module*. Dann muß beim Zugriff auf die Bezeichner lediglich die Abkürzung des Modulnamens angegeben werden, also *m.ident*.

Durch den qualifizierten Import werden Module leichter verständlich, da die Herkunft aller Bezeichner bei jedem Zugriff auf einen Blick sichtbar ist.

In Modula-2 gab es drei verschiedene Arten von Modulen: Definitionsmodule, Implementationsmodule und Hauptmodule. In Oberon gibt es nur noch eine Art von Modul, das die Funktionen aller drei Modultypen in sich vereint. Die exportierten Bezeichner eines Moduls werden einfach mit einem Sternchen hinter ihrem Namen gekennzeichnet. Hauptmodule müssen nicht besonders gekennzeich-

net werden, alle Module können als Hauptmodul dienen und sind ausführbar.

Anweisungen

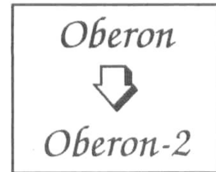
Die WITH-Anweisung hat in Oberon eine völlig andere Aufgabe als sie in Modula-2 hatte (siehe oben). Wie beim Importieren von Bezeichnern müssen auch Recordelementbezeichner immer qualifiziert angesprochen werden.

In Oberon gibt es die FOR-Schleife nicht mehr, da sie leicht durch eine entsprechende WHILE-Schleife ersetzt werden kann. Da man jedoch relativ oft ein solches Schleifenkonstrukt benötigt, wurde die FOR-Schleife bei der Definition von Oberon-2 (Kapitel 13) gleich wieder eingeführt.

Systemnahe Funktionen

Die in Modula-2 möglichen Typkonvertierungen wurden durch die neuen Fähigkeiten von Oberon unnötig. Sie wurden daher in Oberon fallengelassen.

13. Unterschiede zwischen Oberon und Oberon-2



Mit der Sprache Oberon wird seit 1987 gearbeitet. In der Zeit seit der Entwicklung haben sich, besonders für die objektorientierte Programmierung, Wünsche nach einer Erweiterung des Sprachumfangs ergeben. Um diesen Wünschen gerecht zu werden, wurde eine neue Sprache, *Oberon-2*, definiert. Oberon-2 ist eine echte Obermenge von Oberon, das heißt daß jedes korrekte Oberon-Programm auch automatisch ein korrektes Oberon-2-Programm ist.

In dem Bericht [Mössenböck 91] beschreibt Professor Mössenböck von der ETH Zürich die exakten Neuerungen, die in Oberon-2 eingeführt wurden. Auch in [Reiser 92] wurde ein Kapitel Oberon-2 gewidmet.

Die Neuerungen sind schnell aufgezählt: typgebundene Prozeduren, offene Feldtypen, eine erweiterte WITH-Anweisung und die Wiedereinführung der FOR-Schleife.

Die Änderungen im Einzelnen:

Typgebundene Prozeduren

Die Einführung von typgebundenen Prozeduren ist die mit Abstand wichtigste Neuerung in Oberon-2. Sie ermöglicht es, Prozeduren mit RECORD-Typen zu verbinden, und diese dann zu vererben und zu redefinieren, vergleichbar mit Methoden in anderen objektorientierten Sprachen, wie etwa in SmallTalk.

In dem Beispiel des Grafikeditors aus Kapitel 12 würde ein Oberon-2 Programmierer zunächst ein allgemeinen Typ zur Beschreibung von Grafikobjekten schreiben:

```
TYPE
  GrObject = POINTER TO GrObjectDesc;
  GrObjectDesc = RECORD (NodeDesc)
  END;
```

Die Definition dieses Typs erscheint zunächst überflüssig, da das Record außer den Elementen von *NodeDesc* keine neuen Elemente einführt.

Der Oberon-2 Programmierer überlegt sich nun jedoch, welche Eigenschaften ein solches grafisches Objekt haben soll. Allen grafischen Objekten ist gemein, daß sie verschoben, gestreckt, gezeichnet usw. werden können. Diese Eigenschaften können nun mit Hilfe von typgebundenen Prozeduren beschrieben werden. Dies geschieht folgendermaßen:

```
PROCEDURE (go: GrObject) Move(dx,dy: INTEGER);
END Move;

PROCEDURE (go: GrObject) Stretch(f: REAL);
END Stretch;

PROCEDURE (go: GrObject) Draw;
END Draw;

...
```

In runden Klammern vor dem Namen wird jeweils das Ziel der typgebundenen Prozedur angegeben. Dies kann entweder, wie hier, ein Zeiger auf ein Record oder auch ein Record-VAR-Parameter. Mit VAR-Parameter wäre *Draw*:

```
PROCEDURE (VAR g: GrObjectDesc) Draw; END Draw;
```

Es ist kein Fehler, daß die Prozeduren hier keinen Anweisungsteil besitzen, sondern Absicht. Die Deklarationen der Prozeduren sollen

nur anzeigen, daß grafische Objekte diese Prozeduren anbieten. In Erweiterungen werden diese dann durch funktionsfähige Prozeduren ersetzt. Natürlich könnten jedoch auch hier schon richtige Prozeduren definiert werden, daß ist bei diesem Beispiel jedoch nicht sinnvoll.

Wie in Kapitel 12 definieren wir auch hier Recorderweiterungen für Rechtecke und Kreise:

```
TYPE
  Rect = POINTER TO RectDesc;
  RectDesc = RECORD (GrObjectDesc)
    x,y,b,h: INTEGER;
  END;

  Circle = POINTER TO CircleDesc;
  CircleDesc = RECORD (GrObjectDesc)
    mx,my: INTEGER;
    r: INTEGER;
  END;
```

Als Erweiterungen von *GrObjectDesc* erben diese Records nicht nur die Recordelemente von *GrObject*, was in diesem Fall nur die Elemente von *NodeDesc* sind, sondern auch die typgebundenen Prozeduren von *GrObject*.

Um die typgebunden Prozeduren an die neuen Records anzupassen, können sie redefiniert werden. Dies geschieht durch folgende Anweisungen:

```
PROCEDURE (r: Rect) Move(dx,dy: INTEGER);
BEGIN
  INC(r.x,dx);
  INC(r.y,dy);
END Move;

PROCEDURE (r: Rect) Stretch(f: REAL);
BEGIN
```

```
  r.b := SHORT(ENTIER(r.b * f));
  r.h := SHORT(ENTIER(r.h * f));
END Stretch;

PROCEDURE (r: Rect) Draw;
BEGIN
  DrawRect(r.x,r.y,r.b,r.h);
END Draw;

PROCEDURE (c: Circle) Move(dx,dy: INTEGER);
BEGIN
  INC(c.mx,dx);
  INC(c.my,dy);
END Move;

PROCEDURE (c: Circle) Stretch(f: REAL);
BEGIN
  c.r := SHORT(ENTIER(c.r * f));
END Stretch;

PROCEDURE (c: Circle) Draw;
BEGIN
  DrawCircle(c.mx,c.my,c.r);
END Draw;
```

Hier werden in dem selben Modul mehrere gleichnamige typgebundene Prozeduren definiert. Dies ist erlaubt, solange gleichnamige Prozeduren mit unterschiedlichen Recordtypen verbunden sind.

Aufgerufen werden die typgebunden Prozeduren so wie auf Recordelemente zugegriffen wird. Die Variable, die einen Zeiger auf das entsprechende Objekt oder das Objekt selbst enthält, wird gefolgt von einem Punkt und dem Namen der typgebundenen Prozedur. Danach folgt evtl. noch eine Liste der Prozedurparameter. Ist *circle* eine Zeigervariable des Typs *Circle* so ist z.B. der folgende Aufruf möglich:

```
circle.Move(x,y);
```

Beim Aufruf wird die Variable vor dem Namen der typgebundenen Prozedur automatisch als erster Parameter übergeben, bei den oben definierten typgebundenen Prozeduren wird diese Variable also als *r* oder als *c* übergeben.

Eine Prozedur, die alle in einem Baum gespeicherten grafischen Objekte zeichnet, ist nun sehr leicht zu schreiben:

```
PROCEDURE DrawAll(root: GrObject);
BEGIN
  IF root#NIL THEN
    DrawAll(root.left(GrObject));
    root.Draw;
    DrawAll(root.right(GrObject));
  END;
END DrawAll;
```

Der Aufruf *root.Draw* ruft, je nach dem, von welchem Typ das grafische Objekt ist, auf das *root* zeigt, die *Draw*-Prozedur von *Rect* oder die von *Circle* auf. Dies wird als dynamisches Binden bezeichnet. Es ist hier nicht nötig, nach dem Typ der Objekte zu unterscheiden, es reicht zu wissen, daß alle grafischen Objekte gezeichnet werden können.

Sollen weitere grafische Objekte hinzugefügt werden, so kann die Prozedur *DrawAll* unverändert bleiben. Für das neue Objekt müssen nur alle typgebundenen Prozeduren von *GrObject* implementiert werden. Der Rest des Programms, der die typgebundenen Prozeduren benutzt, muß für das neue Objekt nicht angepaßt werden.

Innerhalb der Redefinition einer Prozedur kann die ursprüngliche Prozedur mit einem nachgestellten '^' aufgerufen. So könnte die Prozedur *Draw* von *Circle* oben den Aufruf

```
c.Draw^;
```

enthalten. Dies würde die zu *GrObject* definierte Prozedur *Draw* aufrufen (diese tut jedoch in diesem Beispiel nichts, weshalb hier der Aufruf dieser Prozedur nicht sinnvoll ist.

Offene Feldvariablen

In Oberon-2 ist es möglich, Variablen mit einem Zeiger auf ein offenes ein- oder mehrdimensionales Feld als Typ zu definieren:

```
VAR
  vector: POINTER TO ARRAY OF REAL;
  matrix: POINTER TO ARRAY OF ARRAY OF REAL;
```

Der Speicher für die Felder wird mit der Standardprozedur *NEW* alloziert. Dabei muß nach der Zeigervariablen selbst für jede Dimension des Feldes die Länge in dieser Dimension als Integer-Parameter übergeben werden. Um Speicher für die oberen Variablen zu allozieren, sind z.B. die folgenden Anweisungen möglich:

```
io.WriteString("Vectorlänge?"); io.ReadIntOk(len);
NEW(vector,len);
NEW(matrix,4,4);
```

Die Länge des Feldes kann später mit der Standardfunktion *LEN* erfragt werden:

```
i := LEN(vector^);
REPEAT
  DEC(i);
  vector[i] := 0;
UNTIL i=0;
```

Durch die offenen Felder ist man nicht mehr gezwungen, beim Schreiben eines Programms gleich zu Beginn die Größe von Feldern festzulegen, sondern kann sie flexibel erst zur Laufzeit anpassen. Dies

war in Oberon nur durch das Arbeiten mit Listen und damit mit einem großen Effizienzverlust möglich.

Die FOR-Schleife

In Oberon-2 wurde die FOR-Schleife, die beim Übergang von Modula-2 nach Oberon gestrichen wurde, wieder eingeführt.

Die FOR-Schleife ist eine handliche und leicht zu erkennende Struktur für eine einfache Zählscheife. Solche Schleifen kommen so häufig vor, daß die FOR-Schleife für wichtig genug gehalten wurde, um sie in Oberon-2 wieder einzuführen.

Die FOR-Schleife hat die Form:

```
FOR i := m TO n BY k DO
  StatementSequence
END;
```

Eine gleichwertige WHILE-Schleife kann mit

```
i := m; temp := n;
IF k>0 THEN
  WHILE i<=temp DO
    StatementSequence;
    i := i + k;
  END;
ELSE
  WHILE i>=temp DO
    StatementSequence;
    i := i + k;
  END;
END;
```

beschrieben werden.

Beispiel einer FOR-Schleife:

```
FOR i := 0 TO LEN(vector^-1) DO
  INC(betrag,vector[i]*vector[i]);
END;
betrag := SQRT(betrag);
```

Export nur zum Lesen

Oberon-2 erlaubt es, Variablen und Recordelemente nur zum Lesen zu exportieren. Auf diese Weise ist es für ein Modul möglich, über die Werte von Variablen bestimmte Aussagen zu machen, da die Variablen nur von den Prozeduren dieses Moduls verändert werden können, nicht jedoch von anderen Modulen.

Alle zum freien lesenden und schreibenden Zugriff exportierten Variablen sind auf keine Weise vor Seiteneffekten aus fremden Modulen geschützt.

In Oberon können nur lesbare Variablen simuliert werden, indem sie nicht exportiert werden, stattdessen jedoch eine Prozedur, die den Wert der Variablen als Ergebnis liefert. Dies ist jedoch sehr viel ineffizienter als ein direkter lesender Zugriff.

Zum Lesen exportierte Variablen und Recordelemente werden nicht mit einem Sternchen hinter ihrem Bezeichner markiert, stattdessen bekommen sie an dieser Stelle ein Minuszeichen. Beispiel:

```
VAR
  window - : Display.Window;
TYPE
  Node * = POINTER TO NodeDesc;
  NodeDesc * = RECORD
    left - , right - : Node;
    data * : Entry;
  END;
```

Die WITH-Anweisung

Die Syntax der WITH-Anweisung wurde in Oberon-2 so ergänzt, daß sie ähnlich wie eine CASE-Anweisung angewendet werden kann. Dabei ist die Anweisung

```
WITH p : Rect   DO ...
|    p : Circle DO ...
ELSE ...
END;
```

gleichwertig der folgenden gewöhnlichen Oberon-Anweisung:

```
IF p IS Rect THEN
  WITH p : Rect   DO ...
  END;
ELSIF p IS Circle THEN
  WITH p : Circle DO ...
  END;
ELSE ...
END;
```

Fehlt der ELSE-Teil und schlagen alle Typtests einer Oberon-2 WITH-Anweisung fehl, so wird das Programm mit einem Laufzeitfehler abgebrochen.

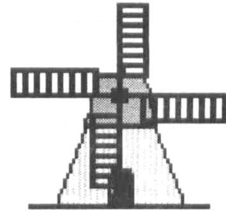
Zusammenfassung

Die wenigen Änderungen, die Oberon-2 von Oberon unterscheiden, machen die Sprache dennoch unwahrscheinlich viel leistungsfähiger.

Die neuen Fähigkeiten ließen sich jedoch alle auch mit etwas Aufwand in Oberon realisieren. Das wichtige an Oberon-2 ist jedoch, daß diese Fähigkeiten nun ein einfach zu verwendender, effizienter Teil der Sprache geworden sind. Nur auf diese Weise wird man auch dazu

ermuntert, sich darüber Gedanken zu machen, ob man einen Typ nicht vielleicht lieber mit flexibel einsetzbaren typgebundenen Prozeduren versieht, als mit starren Prozeduren.

14. Besonderheiten des Compilers



Dieses Kapitel beschreibt die Besonderheiten dieses Compilers. Dazu gehören die Implementationsabhängigen Eigenschaften von Oberon, das Modul *SYSTEM*, die Compileroptionen, verschiedene Speichermodelle und auch Erweiterungen der Sprache.

Besondere Schlüsselwörter und Standardbezeichner

Neue Operatoren AND und NOT

Bei Amiga Oberon ist es möglich, statt den Operatoren '&' und '~' auch die von Modula-2 bekannten Schlüsselwörter *AND* und *NOT* zu verwenden. Dadurch werden boolesche Ausdrücke manchmal übersichtlicher. Für eine bessere Kompatibilität mit andenen Oberon-Systemen sollten jedoch '&' und '~' bevorzugt werden.

Neue Standardbezeichner SHORTSET und LONGSET

SHORTSET und *LONGSET* sind Typbezeichner für Mengen mit 8 bzw. 32 Elementen. Der Mengentyp *SET* kann 16 Elemente enthalten. Beim Programmieren auf dem Amiga ist es oft nötig, Mengentypen in verschieden Größen zur Verfügung zu haben. Daher wurden die zusätzlichen Typen eingeführt.

Mit Variablen das Typs *SHORTSET* und *LONGSET* kann genauso umgegangen werden, wie mit Variablen des Typs *SET*. Diese Mengentypen dürfen jedoch nicht gemischt verwendet werden. Bei der aufzählenden Angabe von Mengen muß bei den neuen Typen der Typbezeichner vor die erste geschweifte Klammer gesetzt werden, damit der Compiler erkennen kann, von welchem Typ die Menge sein soll.

Beispiel:

```
VAR s: LONGSET;
BEGIN
  s := LONGSET{19..29};
END;
```

Neues Schlüsselwort CLOSE

In der ursprünglichen Version des Oberon-Reports [Wirth 87] kannte Oberon hinter dem BEGIN-Anweisungsteil eines Moduls noch einen CLOSE-Teil. Er leitet eine Anweisungsfolge ein, die ausgeführt wird, wenn das Modul geschlossen und aus dem Speicher entfernt wird.

Die CLOSE-Anweisung wird gewöhnlich zum 'aufräumen' benutzt, das heißt, hier werden die Resourcen, die im BEGIN-Teil oder während der Benutzung eines Moduls angefordert wurden, wieder an das System zurückgegeben.

Die CLOSE-Teile werden auch dann ausgeführt, wenn das Programm unkontrolliert, etwa durch einen Benutzerabbruch oder durch einen Laufzeitfehler, abgebrochen wurde.

Beispiel:

```
MODULE abc;
  ...
BEGIN
  window := OpenWindow();
  IF window = NIL THEN HALT(20) END;
CLOSE
  IF window # NIL THEN  CloseWindow(window) END;
END abc.
```

Hier ist sichergestellt, das das Fenster *window* beim Beenden des Programms geschlossen wird.

Neues Schlüsselwort UNTRACED

Die Zeigervariablen eines Oberon-Programms werden gewöhnlich von einem Garbage-Collector verfolgt. Allozierte Objekte, die über verfolgte Zeiger nicht mehr zu erreichen sind, werden automatisch vom Garbage-Collector freigegeben (Kapitel 16).

Manchmal, vor allem beim systemnahen Arbeiten, ist es jedoch nötig, daß der Garbage-Collector einen Zeiger nicht verfolgt. Dies ist z.B. dann der Fall wenn die Zeigervariable auf Speicher zeigen soll, der nicht vom Garbage-Collector alloziert wurde.

Angelehnt an die Notation von Modula-3 [digital 88] können mit Amiga Oberon nicht verfolgte Zeiger definiert werden. Dazu wird bei der Typdefinition vor *POINTER* das neue Schlüsselwort *UNTRACED* gesetzt.

Ein verfolgter Typ ist ein (Record- oder Array-) Typ, der verfolgte Zeiger enthält. Nicht verfolgte Zeiger dürfen nur auf Objekte von nicht verfolgten Typen zeigen, da der Garbage-Collector sonst die verfolgten Zeigervariablen nicht mehr erreichen könnte und ihren Speicher zu früh freigeben könnte.

Nicht verfolgte Zeiger sind nicht kompatibel zu ihren entsprechenden verfolgten Zeigern. Wird für einen nicht verfolgten Zeiger Speicher alloziert, muß dieser explizit freigegeben werden (siehe nächster Abschnitt über *DISPOSE*). Wird der Speicher nicht explizit freigegeben, wird er erst beim Beenden des Programms an das System zurückgegeben.

Beispiel:

```
VAR buffer: UNTRACED POINTER TO ARRAY 500 OF CHAR;
BEGIN
  NEW(buffer); Read(buffer); DISPOSE(buffer);
END;
```

Neue Standardprozedur DISPOSE

Mit dieser Prozedur kann der Speicher, der für einen nicht verfolgten Zeiger alloziert wurde, freigegeben werden.

Wird ein Modul ohne Garbage-Collector compiliert, also mit der Option '-a' oder dem Piktogramm-Merkmal *GARBAGECOLLECTOR=FALSE*, kann mit *DISPOSE* auch der Speicher gewöhnlicher Zeigervariablen freigegeben werden, da dieser dann ja von keinem Garbage-Collector verfolgt wird.

Der Speicher, auf den die Zeigervariable *p* zeigt, wird mit der Anweisung

```
DISPOSE(p);
```

freigegeben.

Die Standardprozedur NEW

NEW liefert, wie von der Sprachdefinition vorgeschrieben, Speicher für das Ziel einer Zeigervariablen. Ist es eine gewöhnliche Zeigervariable, wird Speicher vom Garbage-Collector angefordert, der automatisch freigegeben wird, sobald keine Zeiger mehr auf diesen Speicherblock zeigen.

Ist die übergebene Variable von einem nicht verfolgtem Zeigertyp alloziert *NEW* einen Speicherblock, der vom Garbage-Collector nicht betrachtet wird.

Ist nicht genügend Speicher vorhanden, führt *NEW* zu einem Programmabbruch (siehe unten: *ALLOCATE* aus *SYSTEM*). Wird *NEW* nicht in einem gewöhnlichen Programm, sondern in einer mit LibLink (Kapitel 11) erzeugten Library oder einem Device aufgerufen, so liefert es bei Speichermangel den Wert *NIL*. Hier muß man also besonders vorsichtig sein.

Der Parameter von HALT

Der Parameter der Standardprozedur *HALT* wird als Shell-Rückgabewert verwendet. Er sollte, je nach Schwere des Fehlers, einer der Werte aus dieser Tabelle sein:

```
0    ok
5    Warn
10   Error
20   Fail
```

Wurde das Programm von der Workbench gestartet, so hat der Parameter von *HALT* keine Bedeutung.

Das compilerinterne Module SYSTEM

Dieses Modul enthält systemnahe Funktionen und Typen, die zur direkten Manipulation von Daten ohne die Sicherheit und Kontrolle des Oberon-Typsystems dienen oder den direkten Zugriff auf die Register des Prozessors erlauben.

Ein Definitionsmodul von *SYSTEM* würde in etwa den folgenden Text enthalten (da es keinen Quelltext zu *SYSTEM* gibt, kann es auch kein ordentliches Definitionsmodul geben, der folgende Text dient nur der Veranschaulichung des Moduls):

```
DEFINITION SYSTEM;

TYPE
  BYTE;
  ADDRESS;

PROCEDURE ADR(VAR x: ARRAY OF BYTE): ADDRESS;
PROCEDURE LSH(x: Integer/Set;
              n: INTEGER): Integer/Set;
```

```
PROCEDURE ROT(x: Integer/Set;
              n: INTEGER): Integer/Set;
PROCEDURE SIZE(x: Typ/ARRAY OF BYTE): Integer;
PROCEDURE INLINE(x..: INTEGER);
PROCEDURE REG(n: INTEGER)
                   : LONGINT/ADDRESS/LONGREAL;
PROCEDURE SETREG(n: INTEGER;
                 x: elementarer Typ);
PROCEDURE VAL(T: Typ;
              x: ARRAY OF BYTE): T;
PROCEDURE INIT(p: Pointer Typ);
PROCEDURE TYPEDESC(x: Typ/ARRAY OF BYTE): ADDRESS;
PROCEDURE ALLOCATE(VAR p: Pointer Type);

END SYSTEM.
```

ARRAY OF BYTE steht jeweils für einen Parameter eines beliebigen Typs. *Integer* bezeichnet Parameter der Typen *SHORTINT*, *INTEGER* und *LONGINT*. *Set* steht für Parameter der Mengentypen *SHORTSET*, *SET* und *LONGSET*.

BYTE

Dies ist ein allgemeiner Typ für Werte der Größe ein Byte. Er ist zuweisungskompatibel zu Variablen der Typen *SHORTINT* und *CHAR*. Aus einer Variablen *b* vom Typ *BYTE* kann mit der Anweisung *ORD(b)* ein positiver *INTEGER*-Wert gebildet werden.

Prozedurparametern des Typs *ARRAY OF BYTE* können Werte jedes beliebigen Typs übergeben werden. Dies wird z.B. in den Routinen des Moduls *FileSystem* (Kapitel 22) zum Lesen und Schreiben beliebiger Variablen in Dateien verwendet.

VAR-Parameter des Typs *ARRAY OF BYTE* müssen jedoch mit Vorsicht behandelt werden: Wird ihnen ein Record oder Array übergeben, das verfolgte Zeiger enthält, so können in der Prozedur den verfolgten Zeigern Werte zugewiesen werden, die keine

korrekt allozierten Zeiger sind. Dies führt zu einem inkonsistenten Speichergrafen und damit unweigerlich zu einem Absturz des Garbage-Collectors.

ADDRESS

ADDRESS ist ein allgemeiner Typ für nicht verfolgte Zeigerwerte. Variablen vom Typ *ADDRESS* sind zuweisungskompatibel zu den Typen *LONGINT*, *UNTRACED POINTER TO T* (wobei *T* ein beliebiger Typ, jedoch kein offenes Feld ist) und *BPOINTER TO T* (siehe Kapitel 15).

An Variablen des Typs *ADDRESS* können zudem Werte der Typen *SHORTINT*, *INTEGER*, *LONGSET* und *POINTER TO T* zugewiesen werden.

Die Hauptanwendung des Typs *ADDRESS* ist die Übergabe von Werten an Routinen des AmigaOS. Bei vielen dieser Routinen wird der Typ ihrer Parameter erst aus dem Zusammenhang klar und kann daher nicht starr und für den Compiler prüfbar festgelegt werden.

In Oberon-Programmen sollte man jedoch davon absehen, den Typ *ADDRESS* zu verwenden. Die Sprache Oberon kennt leistungsstarke Mechanismen, die einen solchen Typ in gewöhnlichen Programmen überflüssig machen.

PROCEDURE ADR(VAR x: ARRAY OF BYTE): ADDRESS;

Diese Funktion bestimmt die Speicheradresse der übergebenen Variablen. Ihr können Werte eines beliebigen Typs übergeben werden. *ADR* kann und sollte in gewöhnlichen Oberon-Programmen immer vermieden werden. Durch die Verwendung von *ADR* ergeben sich eine Vielzahl an Fehlermöglichkeiten, die Programme sehr viel unsicherer machen.

Die Hauptanwendung von *ADR* liegt bei der Anwendung mancher Routinen des AmigaOS, die Adressen von Variablen als Parameter verlangen. In vielen Fällen kann mit Prozeduren aus den Interface-Modulen jedoch der Aufruf von *ADR* vermieden werden. Möchte man etwa auf einem Screen zeichnen, benötigt man die Adresse der RastPort-Struktur aus des Screen:

```
Graphics.WritePixel(
            SYSTEM.ADR(screen.rastPort),
            x,y);
```

Mit Hilfe der Routine *ScreenToRastPort* kommt man auch ohne *ADR* aus:

```
Graphics.WritePixel(
            Intuition.ScreenToRastPort(screen),
            x,y);
```

PROCEDURE LSH(x: Integer/Set; n: INTEGER): Integer/Set;

Die interne Darstellung des übergebenen Parameters *x* wird als Binärzahl interpretiert. Ist der Parameter *n* positiv, werden die Ziffern dieser Binärzahl um *n* Positionen nach links, ansonsten um *-n* Positionen nach rechts verschoben. Die freiwerdenden Positionen werden mit Nullen gefüllt. Der Ergebnis dieser Verschiebung wird mit dem Typ von *x* als Funktionsergebnis zurückgegeben.

Beispiel:

```
x := LSH(1,n);
```

Das Ergebnis, das in *x* gespeichert wird, ist hier vom Typ *SHORTINT* und hat den Wert 2^n.

PROCEDURE ROT(x: Integer/Set; n: INTEGER): Integer/Set;

Die interne Darstellung des übergebenen Parameters *x* wird als Binärzahl interpretiert. Ist der Parameter *n* positiv, werden die Ziffern dieser Binärzahl um *n* Positionen nach links, ansonsten um *-n* Positionen nach rechts rotiert. Dabei werden die 'herausfallenden' Ziffern auf der entgegengesetzten Seite wieder eingefügt. Der Ergebnis dieser Rotation wird mit dem Typ von *x* als Funktionsergebnis zurückgegeben.

Beispiel:

```
io.WriteHex(SYSTEM.ROT(12345678H,16),8);
```

Hier wird die Hexadezimalzahl '56781234' ausgegeben.

PROCEDURE SIZE(x: Typ/ARRAY OF BYTE): Integer;

Diese Prozedur ist nur noch aus Kompatibilitätsgründen in *SYSTEM* enthalten. Sie macht nichts anderes als die Standardprozedur *SIZE*, sie gibt die für den übergebenen Typ oder die übergebene Variable benötigten Speicherplatz in Bytes zurück.

PROCEDURE INLINE(x..: INTEGER);

INLINE fügt konstante *INTEGER*-Werte direkt in den Code des Programms ein. *INLINE* hat eine beliebig lange Liste von *INTEGER*-Parametern. Die Parameter werden an der aktuellen Position in das Programm eingefügt.

Die Hauptanwendung in *INLINE* ist das Einfügen von kurzen Maschinensprache-Routinen in Oberon-Programme.

PROCEDURE REG(n: INTEGER)
: LONGINT/ADDRESS/LONGREAL;

REG liefert den Wert des Registers mit der Nummer *n*. Dabei entsprechen die Nummern den folgenden Registern:

```
0  ..  7   Datenregister D0 bis D7
8  .. 15   Adreßregister A0 bis A7
16 .. 23   Fließkommaregister FP0 bis FP7
```

Die Fließkommaregister können nur verwendet werden, wenn Code für einen Fließkommaprozessor (FPU MC68881/2) erzeugt wird, das heißt der Compiler muß mit der Option '-8' oder mit der Piktogramm-Merkmal *MC68881=TRUE* gestartet werden.

Das Ergebnis ist bei Datenregistern vom Typ *LONGINT*, bei Adressregistern vom Typ *ADDRESS* und bei den Registern des Fließkommaprozessors vom Typ *LONGREAL*.

PROCEDURE SETREG(n: INTEGER;x: elementarer Typ);

Der Wert *x* wird in das Register mit der Nummer *n* geschrieben. Die Registernummern sind die selben wie bei der Funktion *REG*. *x* darf dabei kein strukturierter Typ (also kein Record und kein Feld) und nicht *LONGREAL* sein. Nur wenn *n* ein Register des Fließkommaprozessors angiebt, muß *x* vom Typ *LONGREAL* sein.

PROCEDURE VAL(T: Typ;x: ARRAY OF BYTE): T;

Mit dieser Funktion kann das Typsystem von Oberon umgangen werden. Sie dient zum Umwandeln des Typs eines Wertes. Der Typ darf nur mit Vorsicht umgewandelt werden, vor allem beim Ändern von Zeigertypen können leicht Fehler entstehen.

Das Ergebnis ist der Speicherbereich in dem der Wert *x* gespeichert wird so interpretiert, als wäre er vom Typ *T*. Dazu müssen die Größen der Typen, also *SIZE(T)* und *SIZE(x)*, übereinstimmen.

PROCEDURE INIT(p: Pointer Typ);

Mit dieser Routine werden die Zeiger auf die Typdescriptoren (siehe Kapitel 18) der Variablen, auf die *p* zeigt, so gesetzt, daß sie dem statischen Typ von *p* entsprechen.

Ein Aufruf von *INIT* ist dann nötig, wenn auf eine andere Weise als mit *NEW*, etwa mit Exec.AllocMem, Speicher für *p* alloziert wurde und *p^* Records enthält. Beim Allozieren mit *NEW* wird *INIT* automatisch ausgeführt.

PROCEDURE TYPEDESC(x: Typ/ARRAY OF BYTE)
: ADDRESS;

Das Ergebnis ist ein Zeiger auf den Typdescriptor (siehe Kapitel 18) des Typs *x* bzw. des Typs von *x*. Dieser Typ muß ein Record-Typ sein.

PROCEDURE ALLOCATE(VAR p: Pointer Type);

Wie die Standardprozedur *NEW* fordert *ALLOCATE* Speicher für ein neues Objekt an und weist einen Zeiger auf dieses Objekt der übergebenen Zeigervariablen zu. Ist nicht ausreichend Systemspeicher für das gewünschte Objekt vorhanden, so wird hier das Programm jedoch nicht abgebrochen, sondern die Zeigervariable wird mit dem Wert *NIL* belegt. Der Wert der Variablen muß also auf Verschiedenheit von *NIL* geprüft werden, bevor mit dem neuen Objekt gearbeitet werden kann.

Die Compileroptionen

Der Compiler kennt eine große Zahl an Optionen, die sein Verhalten und das des erzeugten Programmcodes beeinflussen. Die meisten Optionen dienen zum Anwählen verschiedener Arten von Überprüfungscode. Andere beeinflussen den erzeugten Code auf verschiedene Arten.

Der Überprüfungscode hilft bei der Suche nach Fehlern in Programmen und verhindert schlimme Folgen von unbemerkten Laufzeitfehlern, wie etwa plötzliche Systemabstürze. Bei gut ausgetesteten Programmen können diese Prüfungen jedoch für eine bessere Effizienz ausgeschaltet werden.

Die meisten Optionen können beim Start des Compilers direkt oder als Piktogramm-Merkmale angegeben werden. Innerhalb des Programmquelltextes können die Optionen in nicht geschachtelten Kommentaren beeinflußt werden. Dabei beginnen sie mit einem Dollarzeichen ($) gefolgt von dem Namen der Option. Das letzte Zeichen gibt an, wie diese Option geändert werden soll. Es muß eines der Zeichen '+', '-' und '=' sein. Sie haben folgende Bedeutung:

```
+    Option wird gesetzt
-    Option wird gelöscht
=    Die Option wird auf den Wert vor der letzten
     Änderung durch '+' oder '-' gesetzt.
```

Um etwa die Überlaufskontrolle für eine Anweisung zu deaktivieren, und danach den vorigen Wert wieder herzustellen, kann folgendes geschrieben werden:

```
(* $OvflChk- *)
  checksum := checksum + x;
(* $OvflChk= *)
```

Wird dieses Modul mit Überlaufskontrolle übersetzt, so schaltet

$OvflChk=* die Überlaufskontrolle wieder ein. Wird es stattdessen ohne Überlaufskontrolle übersetzt, also etwa mit der Option '-v', so läßt *$OvflChk= diese Kontrolle ausgeschaltet.

Die Tabelle unten auf dieser Seite listet alle Optionen auf. Die erste Spalte gibt dabei den Namen der Option an. Groß- und Kleinschreibung wird hier unterschieden. Dahinter steht die Compileroption beim Start von der Shell aus und das entsprechende Piktogramm-Merkmal beim Arbeiten von der Workbench. Nun folgt der voreingestellte Wert der Option. Diesen Wert nimmt die Option an, wenn sie weder beim Compilerstart noch in einem Kommentar angegeben wurde.

Die letzte Spalte zeigt noch an, auf welchen Teil des Moduls sich die Option bezieht: *Module* meint das gesamte Modul. *Proz.* steht für diejenige Prozedur, deren BEGIN-Teil im Text als nächster hinter der Option steht. *stap.* steht für stapelbare Optionen, die wie oben beschrieben, mit '+', '-' und '=' gesetzt werden können. Sie beziehen

Name	Opt	Merkmal	Vorgabe	Bezug
StackChk	-s	STACKCHK	TRUE	stap.
OvflChk	-v	OVFLCHK	TRUE	stap.
RangeChk	-b	RANGECHK	TRUE	stap.
CaseChk	-c	CASECHK	TRUE	stap.
ReturnChk	-r	RETURNCHK	TRUE	stap.
NilChk	-n	NILCHK	TRUE	stap.
OddChk	-o	ODDCHK	FALSE	stap.
TypeChk	-t	TYPECHK	TRUE	stap.
ClearVars	-z	CLEARVARS	TRUE	stap.
Debug	-g	DEBUG	FALSE	stap.
EntryExitCode			TRUE	Proz.
CopyArrays			TRUE	Proz.
SaveRegs			FALSE	Proz.
SaveAllRegs			FALSE	Proz.
DeallocPars			TRUE	Proz.
Implementation			TRUE	Modul
CodeChip			FALSE	Modul
VarsChip			FALSE	Modul
DataChip			FALSE	Modul

sich jeweils auf den gesamten Text hinter der Option bis zum nächsten Vorkommen der Option.

Die exakte Bedeutung der verschiedenen Optionen wird nun beschrieben:

StackChk

Lokale Variablen und Prozedurparameter werden in einem speziellen Speicherbereich, dem sogenannten Stapelspeicher oder Stack, gespeichert. Je nachdem, wie viele Prozeduren aufgerufen werden, wird von diesem Speicher mehr oder weniger benötigt. Auf dem Amiga hat ein Program gewöhnlich nur 4 KByte Stapelspeicher zur Verfügung. Wenn dieser Speicher ausgeht, muß ein Programm abgebrochen werden.

Bei eingeschalteter Stackkontrolle wird nun Code erzeugt, der überprüft, ob immer ausreichend Stapelspeicher vorhanden ist. Geht der Speicher aus, wird das Programm mit einem Laufzeitfehler abgebrochen. Programme, die wenig Stapelspeicher benötigen, können auch ohne Stapelkontrolle übersetzt werden.

Besteht jedoch die Gefahr, daß der Stapelspeicher überläuft, muß mit Stackkontrolle übersetzt werden. Ein Stapelüberlauf bei abgeschalteter Stackkontrolle führt zu einem Systemabsturz. Dieser Fehler kann selbst bei ansonsten völlig korrekten Programmen auftreten.

Bei Prozeduren, die man mit Hilfe des AmigaOS als neue Prozesse oder als Interrupts benutzt, müssen ohne Stackkontrolle übersetzt werden. Dies ist bei Prozessen, die mit dem Modul *Concurrency* (Kapitel 24) gestartet wurden, jedoch nicht nötig.

OvflChk

Die Überlaufskontrolle prüft, ob beim Rechnen mit numerischen

Variablen der Wertebereich dieser Variablen eingehalten wird. In dem Programm

```
MODULE Test;

IMPORT io, NoGuru;

VAR
  i,j: INTEGER;

BEGIN
  i := 15000; j := 20000;
  i := i + j;
  io.WriteInt(i,8);
END Test.
```

führt die Anweisung *i := i + j* zu einem Überlauf, da die Summe *15000 + 20000 = 35000* größer als *MAX(INTEGER)* ist. Mit Überlaufskontrolle übersetzt, führt dieses Programm zu einer Fehlermeldung.

Wird das Programm ohne Überlaufskontrolle übersetzt, so wird der Fehler nicht bemerkt. Das Programm liefert in diesem Fall das (falsche) Ergebnis *-30536*.

RangeChk

Diese Option beeinflußt die Bereichskontrolle. Beim Zugriff auf Feldelemente und die Elemente von Mengen muß geprüft werden, ob der verwendete Index im Bereich des Feldes bzw. der Menge ist. Wird diese Prüfung nicht gemacht, so kann auf nicht existierende Elemente zugegriffen werden, was undefinierte Folgen hat.

Bei schreibendem Zugriff auf Feldelemente mit zu großen Indizes wird beliebiger Speicher überschrieben, was zu bösen Systemabstürzen führt. Die Bereichskontrolle darf daher nur dann

abgeschaltet werden, wenn man sich von der Korrektheit eines Programms überzeugt hat.

CaseChk

Die Case-Index-Kontrolle prüft, ob der Ausdruck einer CASE-Anweisung ohne ELSE-Teil einen der in den Caseindexlisten angegebenen Werte annimmt. Ist dies nicht der Fall wird bei eingeschalteter Case-Index-Kontrolle das Programm abgebrochen.

ReturnChk

Diese Kontrolle prüft, ob jede Funktionsprozedur eine RETURN-Anweisung enthält, die das Resultat der Funktion liefert. Fehlt eine solche Anweisung, wird das Programm abgebrochen. Ist die Return-Kontrolle abgeschaltet, würde die Funktion ein undefiniertes Resultat liefern, was nicht absehbare Folgen hätte.

NilChk

Beim Zugriff auf die Objekte, auf die Zeigervariablen zeigen, wird geprüft, ob die Zeigervariable einen anderen Wert als *NIL* enthält, da *NIL* auf kein Objekt zeigt. Diese Überprüfung heißt NIL-Zeiger-Kontrolle.

Ist sie abgeschaltet, wird bei einem fehlerhaften NIL-Zeiger auf fremden Speicher zugegriffen, was beliebig katastrophale Folgen haben kann.

OddChk

Die Ungerade-Zeiger-Kontrolle ist nur auf Computern sinnvoll, die als Prozessor den MC68020 oder einen neueren Prozessor besitzen. Auf diesen Prozessoren ist es möglich, auf Werte, die größer als ein Byte sind und an ungeraden Speicheradressen

stehen, direkt zuzugreifen. Dies führt auf einem Computer mit einem MC68000-Prozessor zu einem Systemabsturz.

Zugriffe auf ungerade Adressen können nur in Zusammenhang mit nicht verfolgten Zeigern (*UNTRACED POINTER*) bei 'wilden' Zeigermanipulationen mit den Funktionen aus *SYSTEM* auftreten. Da dies in gewöhnlichen Oberon-Programmen nicht getan wird, ist diese Prüfung auch voreingestellt auf *FALSE*.

TypeChk

Beim Zugriff auf erweiterte Records mit der WITH-Anweisung oder bei der Angabe eine Typeguards wird bei der Typkontrolle Code erzeugt, der sicherstellt, daß der Typ des Objekts, auf das zugegriffen wird, auch wirklich der erweiterte Typ ist, sonst wird das Programm mit einem Laufzeitfehler abgebrochen.

Bei abgeschalteter Typkontrolle können fehlerhafte Oberon-Programme auf fremden Speicher zugreifen und damit zu schweren Systemabstürzen führen.

ClearVars

Mit dieser Option wird gewählt, ob die lokalen Variablen von Prozeduren mit Null initialisiert werden sollen (Option gesetzt) oder ob sie zu Beginn jeder Prozedur undefinierte Werte enthalten sollen (Option gelöscht). Siehe hierzu auch den Abschnitt über 'Initialisierung von Variablen' weiter unten in diesem Kapitel.

Debug

Mit dieser Option wird gewählt, ob Code für den Laufzeit-Debugger ODebug (als Zusatzpaket erhältlich, Kapitel 28) erzeugt werden soll. Die Erzeugung von Code für den Debugger kann nur beim Start des Compilers mit der Option '-g' oder dem

Piktogramm-Merkmal *DEBUG=TRUE* eingeschaltet werden, *$Debug+* im Quelltext ist nicht möglich.

Der Sinn der Option *Debug* im Quelltext ist vielmehr, daß man mit ihr abschnittweise die Erzeugung des Debug-Codes abstellen kann, indem man Programmteile mit *(* $Debug- *)* und *(* $Debug= *)* umschließt. Der Grund hierfür kann zum Beispiel sein, daß man sicher ist, daß diese Programmteile korrekt sind, und für eine bessere Effizienz beim Debuggen dort keinen Debug-Code, der das Programm verlangsamt, erzeugen möchte.

Ein anderer Grund kann sich beim Debuggen von sehr großen Modulen dadurch ergeben, daß das Modul mit Debug-Code übersetzt die vorgeschriebene Maximalgröße von 32KByte (siehe unter Einschränkungen in diesem Kapitel) übersteigt. Hier müssen dann evtl. große Teile des Programms zeitweise vom Debuggen ausgenommen werden.

EntryExitCode

Mit dieser Option kann die Erzeugung des Eintritts- und Austrittscodes für eine Prozedur abgestellt werden. Dies ist eigentlich nur dann sinnvoll, wenn man mit *INLINE* aus *SYSTEM* eine Assemblerroutine in ein Oberon-Programm einfügen möchte. Ein Beispiel für eine solche Prozedur enthält das Modul *Strings*:

```
PROCEDURE   Length*(str:   ARRAY  OF   CHAR):
LONGINT;
  (* $EntryExitCode- *)
BEGIN
SYSTEM.INLINE(
  0225FH, 0201FH, 0205FH, 05380H, 02200H,
  04A18H, 057C9H, 0FFFCH, 06708H, 00481H,
  00001H, 00000H, 06AF0H, 09081H, 04ED1H);
END Length;
```

Die Prozedur *Length* wurde aus Effizienzgründen in Assembler

geschrieben. Die Maschinencoderoutine wird hier direkt eingefügt. In gewöhnlichen Oberon-Programmen ist die Verwendung von Assemblerroutinen auf diese Weise jedoch in den meisten Fällen nicht sinnvoll, da der größere Aufwand in keinem Verhältnis zum Geschwindigkeitsgewinn steht.

CopyArrays

Offene Feldparameter müssen in einem relativ aufwendigen Verfahren von einer Prozedur auf den Stapelspeicher kopiert werden. Dies ist besonders dann ärgerlich, wenn die Prozedur eigentlich keinen Wertparameter bräuchte, ein VAR-Parameter jedoch nicht verwendet werden kann, um auch die Übergabe einer konstanten Zeichenkette zu ermöglichen. Hier kann das Kopieren des Feldes mit der Option *$CopyArrays-* verhindert werden.

Eine effiziente Prozedur zur Bestimmung der Länge einer Zeichenkette kann daher folgendermaßen geschrieben werden:

```
PROCEDURE Length(s: ARRAY OF CHAR): LONGINT;
  (* $CopyArrays- *)
VAR l: LONGINT;
BEGIN
  l := 0;
  WHILE (l<LEN(s)) & (s[len]#0X) DO
    INC(l)
  END;
  RETURN l;
END Length;
```

SaveRegs

$SaveRegs+ bewirkt, daß die folgende Prozedur die Prozessorregister D2 bis D7 und A2 bis A7 nicht verändert. Eine solche Prozedur wird z.B. dann benötigt, wenn sie von Routinen des Betriebssystems aufgerufen werden soll.

SaveAllRegs

Ähnlich wie *SaveRegs* rettet eine Prozedur, bei der diese Option gesetzt ist, alle Prozessorregister, also D0 bis D7 und A0 bis A7.

DeallocPars

Anders als in der Programmiersprache C geschriebene Routinen geben Oberon-Prozeduren den Speicher ihrer Parameter selbst frei.

Soll eine Oberon-Prozedur von C-Code, also von C-Programmen oder vom AmigaOS, aufgerufen werden, so darf sie ihre Parameter nicht freigeben. Dies geschieht durch Löschen dieser Option: *$DeallocPars-*.

Implementation

Der Oberon-Compiler erzeugt für jedes Modul eine Objektdatei, selbst wenn das Modul keinen Code enthält, also weder Prozeduren noch einen BEGIN- oder einen CLOSE-Teil besitzt. Ein solches Modul würde nur unnötig Speicher im fertigen Programm verbrauchen.

Durch Löschen von *$Implementation-* wird das Erzeugen von Code und das Speichern einer Objektdatei verhindert.

Bei Modulen, die Recordtypdefinitionen oder exportierte Zeichenkettenkonstanten enthalten, darf diese Option jedoch nicht gelöscht werden, da für den Typdescriptor (Kapitel 18) des Recordtyps und die Konstanten Code erzeugt wird. Die Hauptanwendung dieser Option liegt in den Interfacemodulen zum AmigaOS.

CodeChip

Ist diese Option gesetzt, so wird der Code-Teil dieses Moduls im Chip-Memory gespeichert [RKM: Libraries 92].

VarsChip

Ist diese Option gesetzt, so werden die globalen Variablen dieses Moduls im Chip-Memory gespeichert [RKM: Libraries 92].

DataChip

Ist diese Option gesetzt, so wird der Daten-Teil dieses Moduls im Chip-Memory gespeichert [RKM: Libraries 92].

Bedingte Compilation

Amiga Oberon erlaubt die sogenannte bedingte Compilation. Dadurch wird es möglich, verschiedene Versionen eines Programms oder eines Moduls durch das Setzen oder Löschen verschiedener Optionen beim Start des Compilers zu generieren. Mögliche Anwendungen sind eine Voll- und eine Demo-Version eines Programms oder verschiedensprachige Versionen eines Programms, die denselben Quelltext besitzen.

Für die bedingte Compilation kennt Amiga Oberon sechs Compileroptionen, die wie die gewöhnlichen Optionen in Kommentaren angegeben werden können:

```
$IF <Option>
$IFNOT <Option>
$ELSE
$END
$SET <Option>
$CLEAR <Option>
```

Dabei steht *<Option>* für eine beliebige Zeichenkette, die aus Buch-

staben und Ziffern besteht. Sie gibt den Namen der Option an, hierbei wird Groß- und Kleinschreibung unterschieden. Eine Option mit einem bestimmten Namen kann beim Aufruf des Compilers mit *SET <Name>* gesetzt und mit *CLEAR <Name>* gelöscht werden. Alle Optionen, die beim Aufruf des Compilers nicht angegeben werden, sind automatisch nicht gesetzt. Innerhalb eines nicht geschachtelten Kommentars des Quelltextes kann eine Option mit *$SET <Name>* explizit gesetzt und mit *$CLEAR <Name>* gelöscht werden.

Kommt in einem Oberon-Quelltext in einem nicht geschachtelten Kommentar die Anweisung *$IF <Option>* vor, so werden die folgenden Anweisungen nur dann übersetzt, wenn die Option mit dem Namen *<Option>* gesetzt ist. *$END* beendet die bedingte Codeerzeugung. *$ELSE* kehrt die bedingte Codeerzeugung gerade um, wurde also nach der *$IF*-Anweisung Code erzeugt, wird jetzt keiner erzeugt, andernfalls wird nach dem *$ELSE* Code erzeugt.

$IFNOT <Option>* hat die gleiche Funktion wie *$IF <Option>, hier wird jedoch Code erzeugt, wenn die Option nicht gesetzt ist.

Beispiel:

```
MODULE Test;

IMPORT io;

VAR
  string: ARRAY 80 OF CHAR;

CONST
(* $IF English *)
  hallo = "Hello!";
  nochmal = "once again?";
(* $ELSE *)
  hallo = "Hallihallo!";
  nochmal =  "nochmal?";
(* $END *)
```

```
BEGIN
  REPEAT
    io.WriteString(hallo); io.WriteLn;
    io.WriteString(nochmal); io.ReadString(string);
(* $IFNOT English *)
  UNTIL (string#"ja") & (string#"j");
(* $ELSE *)
  UNTIL (string#"yes") & (string#"y");
(* $END *)
END Test.
```

Von diesem Programm kann nun eine deutsche oder eine englische Version erzeugt werden. Wird der Compiler mit

```
Oberon SET English Test
```

gestartet, erzeugt er eine englische Version. Bei

```
Oberon CLEAR English Test
```

wird dagegen eine deutsche Version erzeugt.

Außer den selbstdefinierten Optionsnamen können bei der bedingten Compilation auch eine Reihe an vordefinierten Optionen verwendet werden. Diese fragen den Zustand verschiedener anderer Compileroptionen ab. Die Vordefinierten Optionen sind:

Name	Opt	Merkmal	Vorgabe
ClearVars	-z	CLEARVARS	TRUE
SmallCode	-m	SMALLCODE	FALSE
SmallData	-d	SMALLDATA	FALSE
MC68010	-1	MC68010	FALSE
MC68020	-2	MC68020	FALSE
MC68030	-3	MC68030	FALSE
FPU	-8	MC68881	FALSE
GarbageCollector	-a	GARBAGECOLLECTOR	TRUE

```
Extensions          -e   EXTENSIONS        TRUE
Debug               -g   DEBUG             FALSE
EntryExitCode                              TRUE
Implementation                             TRUE
```

Der Wert der Optionen hängt von der Position im Quelltext ab, an der sie abgefragt werden.

Beispiel:

```
VAR
(* $IF FPU *)
  x,y: LONGREAL;
(* $ELSE *)
  x,y: REAL;
(* $END *)
```

Hier werden die Variablen *x* und *y* als *LONGREAL* deklariert, wenn das Programm für die Verwendung mit einem Fließkommaprozessor übersetzt wird, ansonsten sind die Variablen *REAL*.

Diese Deklaration ist sinnvoll, da der Fließkommaprozessor bei besserer Genauigkeit mit *LONGREAL*-Werten effizienter rechnet als mit *REALs*. Ohne Fließkommaprozessor sind *REALs* effizienter.

JOIN

Mit *$JOIN <Name>* in einem nicht geschachtelten Kommentar wird dem Compiler mitgelteilt, daß an die erzeugte Objektdatei noch die Objektdatei *<Name>* angehängt werden soll. Diese Objektdatei kann beispielsweise Assembler-Unterroutinen enthalten. Dies wird im nächsten Kapitel genauer beschrieben.

Die unterschiedlichen Speichermodelle

Kurze Programme und Programme mit wenigen globalen Variablen können durch die Verwendung der kleinen Speichermodelle noch weiter optimiert werden. Es muß dem Compiler jedoch explizit mitgeteilt werden, daß er diese Speichermodelle verwenden soll, da die Einschränkungen, die sie mit sich bringen, für große Programme nicht gelten sollen.

Das kleine Code-Modell

Dieses Speichermodell wird durch die Compiler- und Linkeroption '-m' bzw. das Piktogramm-Merkmal *SMALLCODE=TRUE* aktiviert. Bei der Verwendung dieses Moduls werden alle Aufrufe von Prozeduren aus importierten Modulen optimiert. Dies funktioniert jedoch nur, wenn der Code-Teil des Programms insgesamt nicht länger als 32KByte wird.

Wird der Code-Teil dennoch größer als 32KByte, kann OLink meist zusätzlichen Code erzeugen, so daß dennoch ein lauffähiges Programm erzeugt werden kann. Dieses Programm ist dann jedoch gewöhnlich länger und langsamer als es wäre, wenn das große Code-Modell verwendet wird.

Das kleine Daten-Modell

Aktiviert wird dieses Speichermodell durch die Compiler- und Linker-Option '-d' bzw. deren Piktogramm-Merkmal *SMALLDATA=TRUE.* Beim kleinen Datenmodell werden die Speicherbereiche der globalen Variablen aller Module eines Programms zu einem Speicherblock zusammengefügt. Beim Wechsel zwischen zwei Modulen, etwa durch den Aufruf einer importierten Prozedur, muß dann nicht mehr zwischen den beiden Variablenbereichen dieser Module gewechselt werden. Es entfällt hierbei viel Code in jeder Prozedur, wodurch Prozeduraufrufe beschleunigt werden. Außerdem kann im kleinen Daten-

Modell effizienter auf die globalen Variablen importierter Module zugegriffen werden.

Beim kleinen Daten-Modell wird der Speicher der globalen Variablen erst zur Laufzeit alloziert. Dadurch werden Programme, die dieses Speichermodell verwenden reentrant und residentfähig. Sie können also mit dem Shell-Befehl 'resident' speicherresident geladen werden und dann mehrfach gestartet werden. Es wird dann bei jedem Start ein neuer Speicherbereich für die globalen Variablen angelegt.

Die Einschränkung, die das kleine Datenmodell mit sich bringt, wird von den meisten Programmen erfüllt: Die globalen Variablen dürfen zusammen insgesamt maximal 32KByte an Speicher belegen. Enthält das Programm große Felder, so kann das Überschreiten dieser Grenze durch Verwenden von Zeigern auf diese Felder und deren Allozierung zur Laufzeit verhindert werden.

Wird ein Modul mit dem kleinen Daten-Modell compiliert, so müssen auch alle anderen Module des selben Programms mit diesem Speichermodell compiliert werden. Auch beim Linken muß mit der Option '-d' oder dem Piktogramm-Merkmal *SMALLCODE=TRUE* OLink mitgeteilt werden, daß hier ein Programm mit kleinem Daten-Modell erzeugt werden soll.

Um die Objektdateien beim Übersetzen mit dem kleinen Daten-Modell von denen mit großem Daten-Modell unterscheiden zu können, bekommen sie die Endung '.objs'.

Beim kleinen Datenmodell muß beachtet werden, daß bei jedem Zugriff auf globale Variablen das Prozessorregister A5 den Zeiger auf den Bereich der globalen Variablen erhält. Dies ist in normalen Oberon-Programmen immer der Fall. Wird jedoch eine Prozedur mit Hilfe des AmigaOS als neuer Prozess oder als Interrupt gestartet, ist dies jedoch nicht automatisch der Fall. Wird jedoch das Modul *Concurrency* (Kapitel 24) zum Starten von Prozessen verwenden, so braucht man sich darum nicht zu kümmern.

Kombination der Speichermodelle

Es ist natürlich möglich, die beiden kleinen Speichermodelle zu kombinieren um die Vorteile beider Modelle gleichzeitig auszunutzen. Dies ist besonders bei der Programmierung kleiner Shell-Utilities oder anderer Hilfsprogramme, bei denen oft auch die Residentfähigkeit eine Rolle spielt, sinnvoll.

Das compilerinterne Modul MATHLIB

Programme, die viel mit *LONGREAL*-Werten rechnen, können durch die Ausnutzung eines Fließkommaprozessors stark beschleunigt werden. Dies geschieht durch Compilieren mit der Option '-8' bzw. dem Piktogramm-Merkmal *MC68881=TRUE*. Hierbei wird jedoch lediglich für die Grundrechenarten und die Funktionen *ABS* und *ENTIER* der Fließkommaprozessor (im folgenden FPU abgekürzt) ausgenutzt. Die FPUs MC68881 bzw. MC68882 sind jedoch sehr viel leistungsfähiger und unterstützen sogar aufwendige trigonometrische Funktionen.

Damit die Befehle der FPU voll ausgenutzt werden können, kann bei der Codeerzeugung für die FPU ein compilerinternes Modul (ähnlich wie das Modul *SYSTEM*) importiert werden, das genau die Befehle der FPU enthält. Ein Definitionsmodul dieses Moduls würde in etwa folgendermaßen aussehen:

```
DEFINITION MATHLIB;

PROCEDURE ACOS  (x: LONGREAL): LONGREAL;
PROCEDURE ASIN  (x: LONGREAL): LONGREAL;
PROCEDURE ATAN  (x: LONGREAL): LONGREAL;
PROCEDURE ATANH (x: LONGREAL): LONGREAL;
PROCEDURE COS   (x: LONGREAL): LONGREAL;
PROCEDURE COSH  (x: LONGREAL): LONGREAL;
PROCEDURE ETOX  (x: LONGREAL): LONGREAL;
PROCEDURE LOG10 (x: LONGREAL): LONGREAL;
```

```
PROCEDURE LOG2   (x: LONGREAL): LONGREAL;
PROCEDURE LOGN   (x: LONGREAL): LONGREAL;
PROCEDURE SIN    (x: LONGREAL): LONGREAL;
PROCEDURE SINH   (x: LONGREAL): LONGREAL;
PROCEDURE SQR    (x: LONGREAL): LONGREAL;
PROCEDURE SQRT   (x: LONGREAL): LONGREAL;
PROCEDURE TAN    (x: LONGREAL): LONGREAL;
PROCEDURE TANH   (x: LONGREAL): LONGREAL;
PROCEDURE TENTOX(x: LONGREAL): LONGREAL;
PROCEDURE TWOTOX(x: LONGREAL): LONGREAL;
PROCEDURE INT    (x: LONGREAL): LONGREAL;

END MATHLIB.
```

Die Funktionen berechnen dabei die folgenden mathematischen Werte:

SIN, COS, TAN	sin(x), cos(x), tan(x)
ASIN, ACOS, ATAN	arcsin(x), arccos(x), arctan(x)
SINH, COSH, TANH	sinh(x), cosh(x), tanh(x)
ATANH	arctanh(x)
TENTOX	10^x
TWOTOX	2^x
ETOX	e^x
LOG10	$\log_{10}(x)$
LOG2	$\log_2(x)$
LOGN	ln(x)
SQR	x^2
SQRT	$x^{1/2}$
INT	ENTIER(x), für x>=0
	-ENTIER(-x), für x<0

Damit Programme, die mit dem Modul *MATHLIB* geschrieben wurden, auch ohne Verwendung der FPU, und damit auch ohne die effiziente Ausnutzung der Befehle der FPU, übersetzt werden kann, ent-

hält die Modulbibliothek ein gleichnamiges Modul *MATHLIB*, daß diese Funktionen ohne Verwendung der FPU mit Hilfe des AmigaOS implementiert. Wird ein Modul, daß die FPU voll mit dem compilerinternen Modul *MATHLIB* ausnutzt, ohne Ausnutzung der FPU compiliert, so wird also das gleichnamige wirkliche Modul *MATHLIB* der Modulbibliothek verwendet.

Initialisierung von Variablen

Laut der Definition von Oberon haben Variablen, denen noch kein Wert zugewiesen wurde, einen undefinierten Wert. Da es zu unvorhersehbaren und schwer auffindbaren Fehlern führen kann, wenn eine undefinierte Variable gelesen wird, werden von Amiga Oberon alle Variablen je nach ihrem Typ mit den Werten *0*, *NIL*, *0X*, *0.0*, *{}*, *SHORTSET{}*, *LONGSET{}* oder "" vorbelegt.

In portablen Oberon-Programmen, die auch auf anderen Oberon-Compilern auf anderen Computern korrekt übersetzt werden sollen, darf die Vorbelegung der Variablen jedoch nicht vorausgesetzt werden.

Die Vorbelegung der lokalen Variablen der Prozeduren kann mit der Option *ClearVars* beeinflußt werden. Ist sie nicht gesetzt, so sind die Werte der Variablen zu Beginn einer Prozedur undefiniert. Das Abschalten dieser Option macht Programme, die die vorbelegten Werte nicht verwenden, etwas effizienter.

Strukturierte Funktionsergebnisse

Nach dem Oberon-Report [Wirth 90] dürfen Funktionsprozeduren nur einfache Typen als Ergebnis liefern, also keine Records oder Arrays. Es ist jedoch nicht einzusehen, wieso hier zwischen den verschiedenen Typen unterschieden werden soll. Als Argumente sind alle beliebigen Typen erlaubt. Mathematisch gibt es keinen Grund dafür, lediglich bestimmte Typen als Ergebnisse zuzulassen.

In vielen Fällen ist es sogar sehr wünschenswert, komplexere Typen als Funktionsergebnis zu erlauben. So kann ein Modul zum Rechnen mit Vektoren etwa folgendermaßen aussehen:

```
MODULE Vect;

TYPE
  Vector * = ARRAY 3 OF REAL;

PROCEDURE Add(a,b: Vector): Vector;
VAR c: Vector;
BEGIN
  FOR i := 0 TO 2 DO
    c[i] := a[i] + b[i];
  END;
  RETURN c;
END Add;

...

END Vect.
```

Wären strukturierte Ergebnisse nicht möglich, müßte die Prozedur *Add* über den Umweg variable Parameter implementiert werden und würde so ihre logische Anwendungsweise verlieren.

Um drei Vektoren *a*, *b* und *c* zu addieren und in *d* zu speichern, kann nun einfach folgendes geschrieben werden:

```
d := Add(Add(a,b),c);
```

Mit variablen Parametern wäre diese Anweisung ungleich aufwendiger und ihre Funktion weniger einleuchtend.

Designatoren mit Funktionsaufrufen

In Oberon ist es nicht möglich, auf das Ergebnis eines Funktionsaufrufs gleich nach dem Aufruf voll zuzugreifen. Ist das Ergebnis etwa

ein Zeiger auf ein Record oder ein Array, so kann nicht gleich auf die Elemente des Records bzw. des Arrays zugegriffen werden. Dies wird durch die Grammatik von Oberon verboten.

Es ist jedoch nicht einzusehen, weshalb ein solcher Zugriff verboten sein soll. In vielen Fällen kann dadurch eine logischere und kürzere Schreibweise erreicht werden, während der Compiler auch noch effizienteren Code erzeugen kann.

Nehmen wir an, ein Oberon-2-Modul zum Arbeiten mit dreidimensionalen Vektoren stellt folgendes zur Verfügung:

```
DEFINITION Vector3;

TYPE
  Vector = POINTER TO VectorDesc;
  VectorDesc = RECORD
    PROCEDURE (a: Vector) Mul(b: Vector): Vector;
    PROCEDURE (a: Vector) Norm(): REAL;
  END;

END Vector3.
```

a.Mul(b) berechnet das Vektorprodukt der beiden Vektoren *a* und *b*. *a.Norm()* ergibt die Länge von *a*.

Möchte man nun die Fläche *f* des Parallelogramms berechnen, das von zwei solchen Vektoren *a* und *b* aufgespannt wird, genügt folgender Aufruf:

```
f := a.Mul(b).Norm();
```

Wie man sieht, sind solche Aufrufe besonders beim Arbeiten mit Oberon-2 sinnvoll. In gewöhnlichen Oberon-2 müßte das Vektorprodukt in einer Zwischenvariablen gespeichert werden. Als Ziel von Zuweisungen oder als VAR-Parameter dürfen Designatoren auch bei Amiga Oberon keine Funktionsaufrufe enthalten.

Deklarationen

Amiga Oberon erlaubt es, Prozedurdeklarationen mit Konstanten-, Typen- und Variablendeklarationen zu mischen. So entstehen oft übersichtlichere Quelltexte. In gewöhnlichem Oberon müssen die Prozeduren hinter den Deklarationen der globalen Variablen, Typen und Konstanten deklariert werden.

Zeichenkettenkonstanten

Zeichenweiser Zugriff

Amiga Oberon erlaubt den zeichenweisen Zugriff auf Zeichenkettenkonstanten. So ist z.B. die folgende Prozedur möglich:

```
PROCEDURE IntToHex(i: LONGINT;
                   VAR hex: ARRAY OF CHAR;
                   n: INTEGER);
CONST digits = "0123456789ABCDEF";
BEGIN
  hex[n] := 0X;
  WHILE n>0 DO
    DEC(n);
    hex[n] := digits[i MOD 16];
    i := i DIV 16;
  END;
END IntToHex;
```

Diese Prozedur wandelt die positive *INTEGER*-Zahl *i* in eine *n*-stellige Hexadezimalzahl um und speichert sie in der Variablen *hex*.

Steuerzeichen in Zeichenketten

Mit Hilfe des Schrägstrichs rückwärts '\' können in Zeichenketten Steuerzeichen eingefügt werden. Der Schrägstrich rückwärts wird von einem oder mehreren Zeichen gefolgt, die das Steuerzeichen angeben. Es sind folgende Zeichen möglich:

Zeichen	Bedeutung	Wert
\n	line feed	0AX
\t	tabulator	09X
\r	carriage return	0DX
\b	back space	08X
\f	form feed	0CX
\o	nul	00X
\e	escape	1BX
\\	backslash	5CX
\'	single quote	"'"
\"	double quote	'"'
\NNN	Zeichen mit Octalwert NNN	
\xHH	Zeichen mit Hexadezimalwert HH	0HHX
\[	Control Sequence Introducer CSI	9BX

Beispiel: Die Ausgabe von

```
io.WriteString("Hallo!\n");
```

ist dieselbe wie die der Anweisungen

```
io.WriteString("Hallo!");  io.WriteLn;
```

Ein Schrägstrich rückwärts am Ende einer Zeile bewirkt, daß die Zeichenkette am Anfang der nächsten Zeile fortgesetzt wird.

Mehrzeilige Zeichenketten

Manchmal benötigt man sehr lange Zeichenketten, wenn man etwa einen längeren Text auf einmal ausgeben möchte. Dazu können in Oberon einfach mehrere Zeichenketten hintereinander geschrieben werden. Der Compiler macht daraus eine einzige Zeichenkette, indem er alle einzelnen Zeichenketten zusammenhängt. Die einzelnen Zeichenketten können dabei auch über mehrere Zeilen verteilt werden:

```
io.WriteString("Dies ist ein Beispiel\n"
               "für eine sehr lange "
               "Zeichen"  "kette.\nSie erstreckt"
               " sich über 5 Zeilen\nund"
               " 7 "            "Teilstrings\n");
```

Die Ausgabe dieser Anweisung ist der Text:

```
Dies ist ein Beispiel
für eine sehr lange Zeichenkette.
Sie erstreckt sich über 5 Zeilen
und 7 Teilstrings
```

Strukturierte Konstanten

Es ist oft nötig, feste Datenstrukturen, wie etwa Tabellen, in einem Programm zu speichern. Oberon bietet, mit der Ausnahme konstanter Zeichenketten, leider keine Möglichkeit, strukturierte Konstanten zu deklarieren. Man muß sich stattdessen mit Variablen begnügen, denen man zu Programmbeginn feste Werte zuweist.

Dies ist jedoch nicht nur mit viel Tippaufwand verbunden, sondern beeinflußt auch die Programmgröße und die für die Initialisierung benötigte Rechenzeit deutlich.

Amiga Oberon bietet daher die Möglichkeit, auch Konstanten von Record- oder Arraytypen zu bilden. Diese Konstanten werden wie ein Funktionsaufruf geschrieben, wobei statt dem Funktionsbezeichner der Name des Typs verwendet wird. Die Parameter sind die Elemente des Records in derselben Reihenfolge wie in der Typdeklaration oder die Feldelemente in der Reihenfolge aufsteigender Indizes. Bei konstanten erweiterten Recordtypen müssen zunächst die Elemente des Basistyps, danach die der Erweiterung angegeben werden.

Auf so definierte strukturierte Konstanten kann wie auf Variablen zugegriffen werden, ihre Werte können jedoch nicht verändert werden.

Das folgende Beispiel öffnet ein Intuition-Fenster mit einer konstanten *NewWindow*-Struktur. Es zeigt auch gleich ein großes Problem der strukturierten Konstanten: Sie werden schnell sehr unübersichtlich, wenn man sich nicht mit Kommentaren merkt, welche Konstante welchem Recordelement zugewiesen wird.

```
MODULE Test;
IMPORT Int := Intuition;
CONST
  NewWindow = Int.NewWindow(
    0, 0,500,200,          (* dimensions          *)
    1,0,                   (* pens                *)
    LONGSET{},LONGSET{},   (* IDCMP, flags        *)
    NIL,NIL,               (* firstgadg, checkm   *)
    NIL,NIL,NIL,           (* title, screen, bm   *)
    0,0,-1,-1,             (* min/max dims        *)
    {Int.wbenchScreen});   (* type                *)
VAR window: Int.WindowPtr;
BEGIN
  window := Int.OpenWindow(NewWindow);
CLOSE
  IF window#NIL THEN Int.CloseWindow(window) END;
END Test.
```

Operatoren und Spezialsymbole

Der Amiga Oberon Compiler kennt die in der folgenden Tabelle aufgelisteten Operatoren und Spezialsymbole:

"	(	,	/	<	>=	{
#	)	-	:	<=	[	|
&	*	.	:=	=	]	}
'	+	..	;	>	^	~

Reservierte Wörter

Amiga Oberon kennt folgende reservierte Wörter. Sie dürfen nicht als Bezeichnernamen verwendet werden.

AND	ELSE	MOD	STRUCT
ARRAY	ELSIF	MODULE	THEN
BEGIN	END	NOT	TO
BPOINTER	EXIT	OF	TYPE
BY	FOR	OR	UNTIL
CASE	IF	POINTER	UNTRACED
CLOSE	IMPORT	PROCEDURE	VAR
CONST	IN	RECORD	WHILE
DIV	IS	REPEAT	WITH
DO	LOOP	RETURN	

Standardbezeichner

Folgende Bezeichner sind bei Amiga Oberon vordefiniert:

ABS	ENTIER	LONGINT	REAL
ASH	EXCL	LONGREAL	SET
BOOLEAN	FALSE	LONGSET	SHORT
CAP	HALT	MAX	SHORTINT
CHAR	INC	MIN	SHORTSET
CHR	INCL	NEW	SIZE
COPY	INTEGER	NIL	TRUE
DEC	LEN	ODD	
DISPOSE	LONG	ORD	

Anders als in [Wirth 90] vorgeschrieben, ist *NIL* kein reserviertes Wort, sondern ein Standardbezeichner. Dies vereinfacht den Compiler etwas, hat auf das Programmieren in Oberon jedoch keine entscheidenden Auswirkungen.

Wertebereiche

Die Standardtypen können Werte innerhalb der in folgender Tabelle angegebenen Bereiche annehmen:

Typ	MIN(Typ)	MAX(Typ)
SHORTINT	-128	127
INTEGER	-32768	32767
LONGINT	-2147483648	2147483647
REAL	-9.22337177E+18	9.22337177E+18
LONGREAL	-1.7976931348E+308	1.7976931348E+308
SHORTSET	0	7
SET	0	15
LONGSET	0	31
BOOLEAN	FALSE	TRUE
CHAR	0X	0FFX
SYSTEM.BYTE	0X	0FFX

Einschränkungen

Da ein (realer) Computer eine endliche Maschine ist, muß es bei den Programmen, die auf diesem Computer abgearbeitet werden können, gewisse Einschränkungen geben. Diesen Einschränkungen unterliegen natürlich auch der Compiler und die Programme, die von ihm erzeugt werden.

Zudem wurden Einschränkungen gemacht, die keine größeren Nachteile mit sich bringen, die Programme jedoch entscheidend beschleunigen, verkürzen oder vereinfachen.

Die Einschränkungen im Einzelnen:

Codegröße

Der Code-Teil jedes Moduls darf vor der Optimierung maximal 32768 Bytes umfassen. Ein Modul, das diesen Umfang hat,

sollte, allein schon wegen der besseren Übersichtlichkeit, in mehrere Module aufgespalten werden.

Bei Verwendung des kleinen Code-Modells sollte der Code eines Programms insgesamt 32768 Bytes nicht übersteigen (siehe unter 'Die unterschiedlichen Speichermodelle').

Bezeichnernamen

Bezeichnernamen dürfen maximal aus 79 Zeichen bestehen.

Globale Variablen

Es dürfen nicht mehr als 2 GByte an globalen Variablen je Modul verwendet werden. Um effizienter auf die globalen Variablen zugreifen zu können, sollten sie pro Modul nicht mehr als 32768 Bytes belegen.

Bei Verwendung des kleinen Daten-Modells darf ein Programm insgesamt nicht mehr als 32768 Bytes an globalen Variablen besitzen (siehe unter 'Die unterschiedlichen Speichermodelle').

Lokale Variablen und Prozedurparameter

Der innerhalb einer Prozedur für die lokalen Variablen, die Prozedurparameter der Prozedur, die Prozedurparameter von dieser Prozedur aufgerufener Prozeduren und die Prozeduraufrufverwaltung benötigte Speicher darf 32768 Bytes nicht übersteigen.

Der insgesamt von mehreren Prozeduren verwendete Stapelbereich darf jedoch beliebig groß sein.

Symboldateien

Symboldateien dürfen nicht länger als 8388606 Bytes (fast 8 MByte) werden.

Module

Ein Programm darf nicht aus mehr als 32767 Modulen bestehen.

Stapelbare Optionen

Stapelbare Optionen dürfen maximal bis zur Tiefe 64 geschachtelt werden.

Recordtyp-Definitionen

Ein Modul darf nicht mehr als 256 Recordtypen definieren.

Typgebundene Prozeduren

Ein Recordtyp darf nicht mehr als 8191 typgebundene Prozeduren besitzen.

Zeichenkettenkonstanten und strukturierte Konstanten

Die Konstanten dürfen maximal 2 GByte Speicher belegen

Modulname

Der Name eines Moduls darf aus nicht mehr als 25 Zeichen bestehen.

Commands

In [Reiser 92] wird beschrieben, daß Oberon-Programme kein Hauptmodul und keine Hauptroutine besitzen. Stattdessen wird jede exportierte Prozedur als *Command* aufgefaßt, die durch qualifizierte Eingabe ihres Namens aufgerufen werden kann.

Dieses Konzept ist jedoch kaum vereinbar mit dem heutigen Standard von Benutzeroberflächen und bedienerfreundlichen Programmen. So müßte ein Benutzer eines Oberon-Programms dessen *Commands* und deren Parameter kennen, um sie benutzen zu können.

Bei Amiga Oberon wird der BEGIN-Anweisungsteil des beim Linken angegebenen Hauptmoduls für das erzeugte Programm als Hauptanweisungsteil verwendet.

Möchte man auch bei Amiga Oberon mit *Commands* arbeiten, so muß man sich ein übergeordnetes Modul schreiben, daß die *Commands* einliest und die entsprechenden Prozeduren Aufruf:

```
PROCEDURE Commands;

IMPORT io, A, B, ...;

VAR command: ARRAY 256 OF CHAR;

BEGIN
  REPEAT
    io.ReadString(command);
    IF    command = "A.List" THEN A.List
    ELSIF command = "A.Do"   THEN A.Do
    ELSIF command = "B.Type" THEN B.Type
    ...
    END;
  UNTIL command="";
END Comands.
```

Für *Commands* mit Parametern muß dieses Modul natürlich entsprechend erweitert werden. Dabei können die einzelnen Argumente mit den Routinen der Module *Strings* oder *STRING* (siehe Kapitel 20) getrennt werden.

Auf einem Computer mit einer solch leistungsstarken und komfortablen grafischen Benutzeroberfläche wie dem Amiga wirkt die Arbeitsweise mit *Commands* jedoch reichlich steinzeitlich. Ein Amiga-Programm sollte seine Funktionen in Menüs und mit anwählbaren Symbolen anbieten, genaueres hierzu wird in [Style 91] beschrieben.

15. Amigaspezifische Erweiterungen



Im vorigen Kapitel wurden eine Reihe an Compilererweiterungen beschrieben, die einem das Programmieren mit Amiga Oberon erleichtern. Zudem wurde der Sprachumfang um Konstrukte ergänzt, die man speziell für die Programmierung des Amiga, den Zugriff auf das AmigaOS und auf Routinen anderer Sprachen benötigt. Diese Erweiterungen und das Laufzeitsystem werden in diesem Kapitel beschrieben.

Das Laufzeitsystem

Das Laufzeitsystem von Amiga Oberon ist besonders auf Flexibilität, Kürze, Effizienz und Sicherheit ausgelegt. Um diese widersprüchlich klingenden Eigenschaften zu vereinbaren, wurde das Laufzeitsystem in mehrere Teile aufgespalten.

Den wichtigsten Teil des Laufzeitsystems bildet das Modul *OberonLib*. Es übernimmt die beim Programmstart und Programmende nötigen Arbeiten ([AmigaDos 91] und [RKM: Libraries 92]) und stellt Routinen für Arithmetik und Speicherverwaltung zur Verfügung, die von den meisten Oberonprogrammen benötigt werden. Daher wird *OberonLib* von jedem Modul automatisch importiert. Eine Beschreibung der Prozeduren und Variablen von *OberonLib* befindet sich in Kapitel 26.

Laufzeitfehler

OberonLib enthält eine einfache Routine, die Laufzeitfehler abfängt und das Programm nach einem Laufzeitfehler korrekt beendet. Dabei werden zunächst die CLOSE-Anweisungen aller Module ausgeführt, bevor die Kontrolle an das System zurückgegeben wird. Im Fall eines Laufzeitfehlers gibt *OberonLib* keine Fehlermeldung aus. Damit man

den Fehler dennoch bemerkt, wird als Shell-Rückgabewert *-1* zurückgegeben. Wurde das Programm von der Shell gestartet, so führt dies zu der Fehlermeldung 'Unbekannter Befehl'. Dies ist zwar nicht der Grund für das Versagen des Programms, diese Meldung ist jedoch besser als überhaupt keine Fehlermeldung.

Zumindest während der Entwicklungsphase eines Programms sollte das Laufzeitsystem Laufzeitfehler exakter anzeigen. Dies ist auch leicht möglich: Die Module *NoGuru* und *NoGuruRq* (Kapitel 25) enthalten Routinen, die für alle Laufzeitfehler entsprechende Meldungen im Klartext ausgeben. Um diese Module zu benutzen reicht es, sie zu importieren, also ihre Namen in der IMPORT-Liste anzugeben. *NoGuru* gibt die Meldung mit Hilfe des Moduls *io* in ein Console-Fenster ([RKM: Libraries 92]) aus, *NoGuruRq* benutzt ein Dialogfenster, wenn das Programm von der Workbench gestartet wurde, oder gibt die Fehlermeldung in dem Shell-Fenster aus, aus dem das Programm aufgerufen wurde.

So führt das Programm

```
MODULE Test;

IMPORT NoGuru;

VAR
    a,b: INTEGER;

BEGIN
    a := 0; b := 0;
    a := a DIV b;
END Test.
```

zu der Fehlermeldung

```
Guru #0005: Division durch 0
Dx 01E68ADF FFFFFFFF 00004E20 079A25FC
   00000001 00000000 01E60000 00000000
Ax FFFFFFFF FFFFFFFF 07E2FC94 07E306E0
   07E2AF44 07E2FB50 0780E2B8 07E39E34
sr 0004
pc 07E2EC12
<RETURN>
```

Dabei bedeutet:

Guru #0005

Der Laufzeitfehler wurde durch die Exception Nummer 5 des Prozessors ausgelöst ([Williams 89]).

Division durch 0

Dies ist die Ursache des Fehlers.

Dx 01E68ADF ...
Ax FFFFFFFF ...
sr 0004

Hier werden die Inhalte der Prozessorregister D0 bis D7, A0 bis A7 und das Statusregister nach dem Auftreten des Fehlers hexadezimal angezeigt. Durch Betrachten der Werte der Register sind manchmal Rückschlüsse auf die genauere Fehlerursache oder die Fehlerposition im Programm möglich.

pc 07E2EC12

Die angegebene Hexadezimalzahl ist der Wert des Programmzählers an der fehlerhaften Position. Fortgeschrittene Programmierer mit Kenntnissen in Assemblerprogrammierung können mit Hilfsprogrammen wie einem Maschinensprachemonitor

(z.B. mit *mon* von Timo Rossi, auf Fish-Disk 310 [Fish]) die fehlerhafte Position genauer untersuchen.

Abbruch mit <Steuerung> + <C>

Das kleine Laufzeitsystem von *OberonLib* enthält keine Möglichkeit, ein Programm mit der Tastenkombination <Steuerung> (Ctrl) und <C> abzubrechen. Diese Aufgabe wird von den Modulen *Break* und *BreakRq* übernommen (Kapitel 25). Dabei gibt *Break* bei einem Abbruch die Meldung in der Shell die Meldung '*** Break.' aus. *BreakRq* teilt dies dem Benutzer mit einem Dialogfenster mit.

Um diese Module zu benutzen reicht es auch hier, sie einfach zu importieren.

Das Prüfen des Programmabbruchs ist mit der Stackkontrolle gekoppelt, so daß nur Programme, die mit dieser Kontrolle übersetzt wurden, abgebrochen werden können. Zudem wird ein Abbruch nur bei Prozeduraufrufen geprüft, so daß z.B. eine endlos laufende Schleife nicht abgebrochen werden kann, wenn sie keinen Prozeduraufruf enthält. Einen Abbruch bei jedem Schleifendurchlauf zu prüfen würde die Effizienz des Programms deutlich verschlechtern. Man kann die Abbruchsprüfung in einer Schleife jedoch explizit angeben, indem man in der Schleife die Prozedur *CheckBreak* aus *Break* bzw. *BreakRq* aufruft.

Bespiel: Das Programm

```
MODULE Endlos;

IMPORT io, Break;

BEGIN
  LOOP
    io.WriteString("Hallo! ");
  END;
END Endlos.
```

kann nur mit der Tastenkombination <Steuerung> und <C> beendet werden.

Der Typ STRUCT

Records in Oberon enthalten immer ein unsichtbares zusätzliches Element, daß Informationen über den Typ des Records enthält und beim Aufruf von typgebundenen Prozeduren des Records verwendet wird (Kapitel 18). Die in den Datenstrukturen des AmigaOS verwendeten Typen enthalten eine solche Zusatzinformation jedoch nicht. Um diese Datenstrukturen in Oberon nachzubilden ist daher ein neuer Typ ähnlich des Records nötig, der jedoch keine versteckten Informationen enthält.

Dieser Typ wird mit dem neuen Schlüsselwort *STRUCT* gebildet. Er wird ansonsten sehr ähnlich wie Recordtypen aufgebaut. Das Schnittstellenmodul *Graphics* enthält beispielsweise folgende Definition

```
TYPE
  RasInfo * = STRUCT
    next * : RasInfoPtr;
    bitMap * : BitMapPtr;
    rxOffset * , ryOffset * : INTEGER;
  END;
```

Da die Datenstrukturen des AmigaOS hierarchisch aufgebaut sind, und viele Strukturen andere enthalten, ähnlich wie erweiterte Records ihre Basisrecords enthalten, können auch mit *STRUCT* definierte Typen erweitert werden. Mit einem Typeguard kann auch hier auf die Elemente der erweiterten Typen zugegriffen werden. Da ein Struct keine Typinformationen enthält, kann der Compiler die Korrektheit des Typguards nicht sicherstellen, dies muß der Programmierer tun.

Als Basistyp eines erweiterten Structs kann nicht einfach ein Typbezeichner in runden Klammern angegeben werden, sondern es muß das erste Element des Structs als Basistyp dienen. Das Interfacemodul *In-*

tuition enthält z.B. folgende Definition eines erweiterten Structs:

```
TYPE
   IntuiMessage * = STRUCT
    (execMessage * : Exec.Message)
     class * : LONGSET;
     code * : INTEGER;
     ...
   END;
```

IntuiMessage wird also als Erweiterung einer Exec.Message definiert.

In einem Programm kann dies folgendermaßen ausgenutzt werden:

```
MODULE Demo;
IMPORT Exec, I := Intuition;
VAR
   msg: Exec.MessagePtr;
   window: Int.WindowPtr;
BEGIN
  ...
  msg := Exec.GetMsg(window.userPort);
  IF msg # NIL THEN
    IF I.rawKey IN I.msg(I.IntuiMessage).class THEN
      ...
    END;
    Exec.ReplyMsg(msg);
  END;
  ...
END Demo.
```

Da die Message *msg* von dem Port eines Fensters kommt, können wir hier sicher sein, daß es sich um eine Message vom Typ *IntuiMessage* handelt, so daß der Typ gefahrlos mit einem Typeguard geändert werden kann.

Der Typ Struct sollte in reinen Oberon-Programmen nicht verwendet werden, da Oberon den leistungsfähigeren und sichereren Typ Record kennt. Die geringe Speicherplatzersparnis bei Structs rechtfertigt ihre

Verwendung nicht. Auch können an Structs keine Prozeduren gebunden werden.

Der Typ BPOINTER

Teile der 'dos.library' des AmigaOS wurden in der Sprache BCPL geschrieben ([AmigaDos 91]). In dieser Sprache werden Zeiger anders gespeichert, als es sonst im Amiga üblich ist. Diese Zeiger müssen daher besonders behandelt werden.

Das neue Schlüsselwort *BPOINTER* kann wie *POINTER* zur Definition von Zeigertypen verwendet werden. Die so definierten Zeiger werden nicht vom Garbage-Collector verfolgt, sie müssen also auf nicht verfolgte Objekte zeigen.

Variablen des Typs *BPOINTER TO T* haben folgende Eigenschaften:

- Auf *T* kann mit '#', '.' oder '[' zugegriffen werden. Dazu wird die Zeigervariable zunächst in einen gewöhnlichen Zeiger umgewandelt, der dann dereferenziert wird.

- Wird die Variable an eine Variable des Typs *UNTRACED POINTER TO T* oder *ADDRESS* zugewiesen, wird ihr Wert zuvor in einen gewöhnlichen Zeiger umgewandelt. Diese Automatische Umrechnung kann, z.B. bei Tag-Listen, auch zu Problemen führen: Gegebenenfalls muß die Umrechnung mit *VAL* aus *SYSTEM* unterdrückt werden.

- Wird ihr der Wert einer Variablen des Typs *UNTRACED POINTER TO T* oder *ADDRESS* zugewiesen, so wird der Wert zunächst in einen BCPL-Zeiger umgewandelt.

Ein *BPOINTER* darf nicht auf einen Recordtyp zeigen. Da mit einem *BPOINTER* nur sehr viel ineffizienter gearbeitet werden kann als mit einem gewöhnlichen Zeiger, sollten *BPOINTER* nur dort eingesetzt werden, wo sie dringend benötigt werden. Dies ist nur beim Zugriff auf die Strukturen der Dos-Library der Fall.

Registerparameter

Manchmal ist es nötig, Prozedurparameter nicht über den Stapelspeicher sondern direkt über Prozessorregister an die Prozedur zu übergeben. Um dies zu bewirken muß die Nummer des Registers (Kapitel 14, Seite 10) bei der Prozedurdeklaration in geschweiften Klammern hinter den Namen des Parameters geschrieben werden.

Oberon-Prozeduren dürfen die Register D0 bis D7 und A0 bis A4 für Registerparameter benutzen. Ist ein Parameter einer Prozedur ein Registerparameter, so müssen alle anderen Parameter auch Registerparameter sein. Parameter in den Registern D0 und D1 sind innerhalb der Prozedur nicht vor dem Überschreiben geschützt, es kann also sein, daß sie schon kurz nach dem Start der Prozedur überschrieben werden.

Registerparameter dürfen nur für elementare Typen verwendet werden, die klein genug sind, um in dem angegebenen Register gespeichert werden können, lediglich bei Library-Prozeduren (s.u.) sind in Adressregistern auch Strukturen möglich, wobei dann automatisch die Adresse übergeben wird. Für numerische Werte dürfen keine Adressregister verwendet werden. Für Zeiger sollten dagegen nur Adressregister verwendet werden.

Registerparameter sind vor allem beim Schreiben von Libraries und Devices (siehe dazu das Kapitel 11 über LibLink) wichtig. Ein Beispiel für eine Prozedur mit Registerparametern in D2 und D3 ist

```
PROCEDURE GGT*(x{2},y{3}: LONGINT): LONGINT;
  (* $SaveRegs+ *)
VAR z: LONGINT;
BEGIN
  IF x>y THEN z := x; x := y; y := z END;
  REPEAT
    z := x; x := y MOD x; y := z;
  UNTIL x=0;
  RETURN y;
END GGT;
```

Die Prozedur berechnet den größten gemeinsamen Teiler der beiden übergebenen Werte.

Aufruf von Libraryroutinen

Die Routinen des AmigaOS sind in Bibliotheken, den Libraries, angeordnet ([RKM: Libraries]). Jede Library hat eine Basisadresse, die man beim ihrem Öffnen mit *OpenLibrary* aus *Exec* enthält. Jede Routine hat einen festen Offset relativ zu dieser Basisadresse.

Um in Oberon direkt die Routinen einer Library aufrufen zu können, können Prozeduren als Libraryprozeduren deklariert werden. Wird eine solche Prozedur aufgerufen, erzeugt der Compiler Code für den Aufruf der entsprechenden Libraryroutine.

Hinter dem Namen und der Exportmarkierung einer Libraryprozedur müssen in geschweiften Klammern eine nicht importierte globale Variable, die die Basisadresse der Library enthält, und der Offset der Routine angegeben werden.

Die Deklarationen der Libraryprozeduren des AmigaOS sind in den Interfacemodulen enthalten. So steht in *Graphics*:

```
PROCEDURE TextLength *{gfx,- 54}(
              rp     {9} : RastPortPtr;
              string{8}  : ARRAY OF CHAR;
              count {0}  : LONGINT): INTEGER;
```

Die Basisadresse steht hier in der Variablen *gfx*. Der Offset von *TextLength* ist *-54*. Es ist zu beachten daß Libraryroutinen gewöhnlich Register für die Übergabe ihrer Parameter verwenden. In Adressregistern sind hier auch Strukturen möglich, der Compiler übergibt beim Aufruf hier automatisch die Adresse der Struktur. So sind auch offene Felder möglich, was hier beim Paramter *string* ausgenutzt wird, dem beliebige Zeichenketten übergeben werden können.

Listenparameter

Manche Betriebssystemroutinen erlauben die Übergabe einer beliebig langen Liste von Parametern. Damit diese Prozeduren auch einfach von Oberon-Programmen aus benutzt werden können, wurde die Sprache leicht erweitert.

Der letzte Parameter eine Prozedur kann mit '..' gekennzeichnet werden, um anzudeuten, daß er beliebig oft (auch null mal) angegeben werden kann. Der Parameter muß dabei ein Registerparameter sein.

Beispiele für solche Routinen sind *PrintF* aus *Dos* oder *OpenWindow-TagsA* aus *Intuition*:

```
PROCEDURE PrintF              *{dos,-954}(
              format{1}       : ARRAY OF CHAR;
              arg1{2}..       : e.APTR);

PROCEDURE OpenWindowTagsA *{int,-606}(
              newWindow{8}        : NewWindow;
              tag1{9}..           : u.Tag): WindowPtr;
```

Aufgerufen werden diese Routinen z.B. mit

```
Dos.PrintF("a=%ld b=%ld c=%ld\n",a,b,c);
win := Int.OpenWindowTags(newwindow,
           Int.waWidth, 500,
           Int.waHeight,200,
           Int.waTitle, SYSTEM.ADR("ListenDemo"),
           Utility.end);
```

In Oberon geschriebene Prozeduren dürfen keine Listenparameter besitzen, da die Sprache Oberon keine Mittel zur Verfügung stellt, auf die Liste zuzugreifen. Stattdessen kennt Oberon offene Feldparameter, mit denen ähnliches erreicht werden kann.

Variablen an absoluten Adressen

Auf dem Amiga gibt es lediglich eine absolute Adresse, nämlich die Basisadresse der Exec-Library die an der Adresse 4 steht.

Variablen in einem Oberon-Programm können an einer absoluten Adresse stehen, wenn diese in eckigen Klammern hinter dem Bezeichner in der Variablendeklaration angegeben wird. So enthält das Interfacemodule zu *Exec*:

```
VAR
  exec *  [4H]: ExecBasePtr;
```

Eine weitere Anwendungsmöglichkeit der Variablen an absoluten Adressen ist der direkte Zugriff auf Hardware-Register (die Amiga-Programmierrichtlinien verbieten dies!). In den meisten Fällen ist der Weg über die Amiga-Devices [RKM: Devices 92]) vorzuziehen.

Variablen und Prozeduren an einem externen Label

Ein Label ist eine durch einen bestimmten Namen ausgezeichnete Adresse einer Variablen, Konstanten oder Routine in einer Objektdatei. Über Labels kann man auf die Daten in einer Objektdatei zugreifen, wenn diese z.B. von einem Compiler einer anderen Programmiersprache oder von einem Assembler stammt.

Um auf eine externe Variable an einem bestimmten Label zugreifen zu können, muß im Oberon-Programm eine Variable deklariert werden. Hinter dem Variablenbezeichner muß dabei der Name des Labels in eckigen Klammern (und in Anführungszeichen) angegeben werden.

Entsprechend kann eine Prozedur an einem externen Label definiert werden: Hinter dem Prozedurbezeichner wird in geschweiften Klammern (und in Anführungszeichen) der Name des Labels angegeben.

Beispiel:

Folgende Assemblerroutine "GGT" berechnet den größten gemeinsamen Teiler zweier positiver *LONGINT*-Zahlen, die in den Registern D0 und D1 übergeben werden. Das Ergebnis wird im Register D0 zurückgeliefert. In der *INTEGER*-Variablen an dem Label "cnt" wird die Anzahl der Schleifendurchläufe gezählt.

```
        XDEF    cnt
        XDEF    GGT

        SECTION "GGT_BSS",BSS

cnt:    dc.w    0

        SECTION "GGT_CODE",CODE

GGT:    clr.w   cnt
loop:   add.w   #1,cnt
        sub.l   D0,D1
        beq.s   done
        bpl.s   loop
        neg.l   D1
        sub.l   D1,D0
        bra.s   loop
done:   rts

        END
```

Dieser Assemblerquelltext kann zum Beispiel mit dem Assembler *a68k* von Charlie Gibbs (auf Fish-Disk 521 [Fish]) durch den Aufruf *a68k ggt.s* in die Objektdatei *ggt.o* übersetzt werden.

In einem Oberon-Programm können die Variable *cnt* und die Routine *GGT* nun verwendet werden, indem zunächst die Variable und die Prozedur an den externen Labels definiert werden. Im Programm können sie dann so benutzt werden, als wäre die Variable eine gewöhnliche Variable und die Prozedur eine Oberon-Prozedur.

```
MODULE GGTDemo;

IMPORT io;

VAR
  zaehler["cnt"]: INTEGER;
  a,b: LONGINT;

PROCEDURE ggt{"GGT"}(a{0},b{1}: LONGINT) :LONGINT;

BEGIN
  REPEAT
    io.WriteString("a = ");
  UNTIL io.ReadInt(a) & (a>0);
  REPEAT
    io.WriteString("b = ");
  UNTIL io.ReadInt(b) & (b>0);
  io.WriteString("GGT(a,b) = ");
  io.WriteInt(ggt(a,b),8); io.WriteLn;
  io.WriteString("zaehler = ");
  io.WriteInt(zaehler, 8); io.WriteLn;
END GGTDemo.
```

Beim Linken dieses Programms mit OLink muß die Objektdatei des Assemblerteils nun zusätzlich angegeben werden. OLink muß also folgendermaßen aufgerufen werden:

```
OLink GGTDemo OBJ GGT.o
```

Eine Möglichkeit, Assemblerteile automatisch von OLink einbinden zu lassen, ist das Anhängen der Objektdatei des Assemblerteils an die eines Oberonmoduls. So würde man in diesem Beispiel ein Modul schreiben, daß die Variable *cnt* und die Prozedur *GGT* deklariert und exportiert:

```
MODULE GGT;

VAR
   cnt * ["cnt"]: INTEGER;

PROCEDURE GGT * {"GGT"}(
               a{0},b{1}: LONGINT):LONGINT;

END GGT.
```

Dieses Modul wird zunächst wie gewohnt compiliert. Danach werden die Objektdateien mit den folgenden Shell-Anweisungen zusammengefügt:

```
> JOIN GGT.obj GGT.o AS T:GGT.obj
> COPY T:GGT.obj TO GGT.obj
```

Danach enthält die Objektdatei *GGT.obj* auch den Assemblerteil. Alle Module, die *cnt* oder *GGT* aus diesem Modul importieren, brauchen nun nicht mehr mit 'OBJ GGT.o' gelinkt werden, da OLink die Objektdatei *GGT.obj* findet und darin der Assemblerteil enthalten ist.

Damit nicht nach jeder Compilation die Objektdateien explizit zusammengefügt werden müssen, können im Quelltext des Moduls mit der Option *$JOIN <Objektdatei>* die Namen der zuzufügenden Objektdateien angegeben werden. Der Compiler liest die angegebenen Objektdateien dann automatisch ein und fügt sie hinten an die erzeugte Objektdatei an. Das Programm oben sollte also um den Kommentar

```
(* $JOIN GGT.o *)
```

ergänzt werden, damit die Datei 'GGT.o' automatisch an die erzeugte Objektdatei angefügt wird.

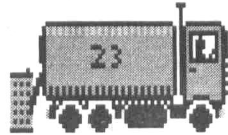
Entsprechend wie hier auf einen Programmteil in Assembler zugegriffen wurde, können auch Programmteile anderer Sprachen von Oberon aus verwendet werden. Manchmal ist hier jedoch mehr Aufwand nötig. Schlimmstenfalls muß eine kurze Assemblerroutine geschrieben werden, die der Routine der anderen Sprache die Parameter korrekt übergibt und sie aufruft.

Vorzeichenlose Hexadezimalzahlen

Im Hexadezimalsystem angegebene Konstanten sollen manchmal ohne Berücksichtigung ihres Vorzeichens eingegeben werden. Daher erlaubt Amiga Oberon die Angabe von Hexadezimalzahlen, die größer als *MAX(LONGINT)* sind. Erlaubt sind die Konstanten im Bereich von *0H* bis *0FFFFFFFFH*. Dabei hat die Konstante *0FFFFFFFFH* den Wert *-1*. Der Wert von *7FFFFFFFH* ist *2147483647*, der von *80000000H* dagegen *-2147483648*.

Manchmal ist auch die Angabe von vorzeichenlosen Hexadezimalzahlen des Typs *INTEGER* sinnvoll, beispielsweise beim Einfügen von Assemblercode mit der Prozedur *INLINE* aus *SYSTEM*. Diese Konstanten können mit einem 'U' statt dem 'H' am Schluß eingegeben werden. Die zulässigen Zahlen reichen von *0U* bis *0FFFFU*. So ist der Wert von *0FFFFU* gleich *-1*, der von *0CE86U* ist *-12666*.

16. Der Garbage-Collector



Ein Garbage-Collector ist ein Programm oder ein Programmteil, das den Speicher, den ein anderes Programm oder ein anderer Programmteil angefordert hat, nach nicht mehr benötigten Speicherbereichen durchsucht und diese an das System zurückgibt.

Wieso Garbage-Collection?

Viele herkömmliche Programmiersprachen, wie etwa 'C' oder Modula-2, haben eine flexible Speicherverwaltung, benutzen jedoch keinen Garbage-Collector. Eine solche Speicherverwaltung bietet wichtige Vorteile gegenüber eines starren Speichersystems, die heutzutage nicht mehr wegzudenken sind.

Dennoch bringt die flexible Speicherverwaltung Gefahren mit sich: Zum einen kann es leicht passieren, daß man vergißt, Speicher an das System zurückzugeben. Unerreichbarer Speicher, der nicht freigegeben wird, wird als Garbage bezeichnet. Programme, die viel Garbage erzeugen, machen einen Computer mit einem begrenzten Speicherbereich auf Dauer unbenutztbar.

Ein schwerer Fehler ist jedoch der Umgekehrte: Wird Speicher freigegeben, auf den noch Zeiger existieren, entstehen sogenannte hängende Referenzen. Wird mit diesen Zeigern weitergearbeitet, wird unkontrolliert Speicher überschrieben. Dies hat katastrophale Folgen: Daten ändern ohne erkennbaren Grund ihre Werte, Programme verursachen in unregelmäßigen Abständen Systemabstürze usw.

Die Idee hinter der Garbage-Collection ist nun, daß zeitweise auftretender Garbage durchaus geduldet werden kann, es wird dadurch ja lediglich wenig Speicher unnötigerweise als belegt angesehen. Hängende Referenzen darf es dagegen nicht geben. Programmiersprachen, die ihren Speicher durch einen Garbage-Collector verwalten, haben daher

keine Anweisung zum Freigeben von Speicher. Es existiert lediglich eine Anweisung zum Anfordern von Speicher, in Oberon ist dies die Standardprozedur *NEW.*

Die Aufgabe eines bestimmten Programmteils, nämlich des Garbage-Collectors, ist es nun, den entstehenden Garbage zu finden und freizugeben. Auf diese Weise erhält man eine Speicherverwaltung, die sicher ist: es können keine hängenden Referenzen entstehen und Garbage wird automatisch freigegeben.

Der Garbage-Collector von Amiga Oberon

Der Garbage-Collector von Amiga Oberon arbeitet parallel zum eigentlichen Programm als eigener Prozeß. Auf diese Weise bemerkt ein Oberon-Programm die Arbeit des Collectors gewöhnlich nicht. Damit mehrere Programme gemeinsam den gleichen Garbage-Collector verwenden können, ist er in einer Amiga-Library, der 'garbagecollector.library', enthalten. Oberonprogramme, die mit verfolgten Zeigern arbeiten und bei eingeschaltetem Garbage-Collector übersetzt wurden, also ohne Angabe der Compileroption '-a' oder mit dem Piktogramm-Merkmal *GARBAGECOLLECTOR=TRUE*, benötigen diese Library, um ausgeführt zu werden.

Aus diesem Grund darf die 'garbagecollector.library' zusammen mit Programmen, die mit Amiga Oberon übersetzt wurden, weitergegeben und sogar kommerziell vertrieben werden (s.u.). Sie muß in das 'LIBS:'-Verzeichnis eines Rechners kopiert werden, auf dem diese Oberon-Programme ausgeführt werden sollen.

Funktionsweise des Garbage-Collectors

Der von diesem Garbage-Collector verwendete Algorithmus ist eine Abwandlung und Erweiterung des in [Dijkstra 78] beschriebenen parallelen Algorithmus.

Der Garbage-Collector startet einen Prozeß niedriger Priorität, den sogenannten Collector-Prozeß oder einfach Collector. Dieser untersucht den Speicher, während die Oberon-Programme, die den Garbage-Collector benutzen, die sogenannten Mutatoren, Speicherbereiche, im folgenden Objekte genannt, anfordern und mit ihnen arbeiten.

Der Collector durchsucht in einem zyklischen Vorgang den gesamten Speicher nach unerreichbaren, also freien Objekten und gibt diese zum Schluß jedes Zyklus frei. Währenddessen arbeiten die Mutatoren weiter, verändern den Zeiger auf Objekte und fordern neue Objekte an. Damit der Collector funktionieren kann, müssen die Mutatoren Objekte, die von Veränderungen betroffen sind, markieren. Der für diese Markierungen erforderliche Programmcode wird vom Compiler automatisch erzeugt.

Benötigt ein Mutator sehr viel Rechenzeit oder sehr viel Speicher, so kann es vorkommen, daß der Collector weniger Objekte freigeben kann, als der Mutator anfordert. Geht der Systemspeicher aus, so wird der Mutator solange angehalten, bis der Collector den nächsten Zyklus beendet und somit (hoffentlich) wieder Speicher freigegeben hat. Je nach der Speichermenge, die der Collector untersuchen muß, kann dies Sekundenbruchteile oder mehrere Sekunden dauern. Erst wenn der Speicher auch nach einem kompletten Zyklus des Collectors nicht ausreicht, wird das Programm des Mutators beendet.

Damit sichergestellt werden kann, daß der Collector ausreichend Rechenzeit zum Auffinden und Freigeben unerreichbarer Speicherbereiche erhält, kann mit dem Voreinstellungs-Editor GarbagePrefs (im nächsten Kapitel beschrieben) eine Maximalzahl an Objekten und eine Höchstmenge an Speicher eingestellt werden, die ein Mutator anfordern kann, während der Collector einen Zyklus bearbeitet. Kann der Collector keinen Zyklus beenden, während der Mutator die angegebene Menge an Speicher anfordert, so wird der Mutator solange angehalten, bis der Collector einen Zyklus beendet hat. So wird verhindert, daß der Mutator den gesamten Systemspeicher anfordert, während der Collector keine Rechenzeit zum Freigeben nicht mehr benötigter Objekte bekommt.

Um das Anfordern und Freigeben von kleinen Objekten zu beschleunigen, verwaltet der Garbage-Collector kleine Objekte selbst und gibt im Augenblick nicht benötigte kleine Objekte nicht sofort an das System zurück. Die maximale Größe und maximale Anzahl der so gespeicherten Objekte kann mit GarbagePrefs eingestellt werden.

Compilation ohne Garbage-Collector

Mit der Compileroption '-a' und dem Piktogramm-Merkmal *GARBAGECOLLECTOR=FALSE* kann ein Programm ohne Verwendung des Garbage-Collectors übersetzt werden. In einem solchen Programm werden alle Zeigervariablen ähnlich wie nicht verfolgte Zeiger behandelt.

Der mit *NEW* angeforderte Speicher kann hier nicht automatisch freigegeben werden. Das Programm gibt seinen Speicher erst dann an das System zurück, wenn es beendet wird. Dies ist natürlich kein wünschenswertes Verhalten.

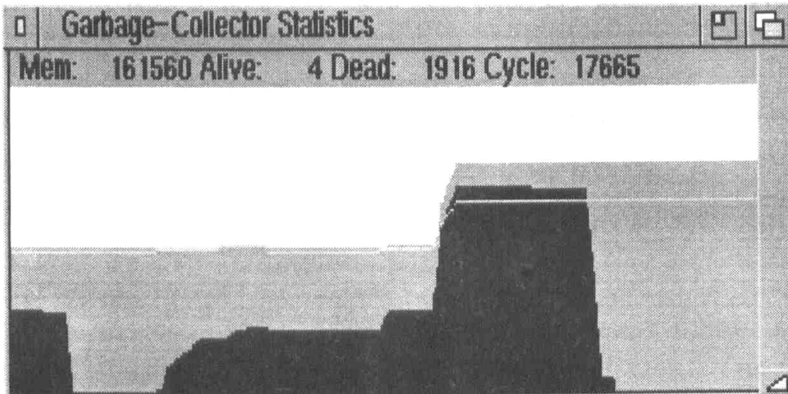
Daher ist es bei der Compilation ohne Garbage-Collector möglich, mit *NEW* angeforderte Objekte mit der Standardprozedur *DISPOSE* explizit freizugeben. Um nun ein Modul zu schreiben, daß sowohl mit als auch ohne Garbage-Collector übersetzt werden kann, bietet es sich an, Objekte mit bedingter Compilation bei Abfrage der Option *Garbage-Collector* freizugeben.

Die folgende Prozedur löscht eine komplette Liste (aus dem Modul Lists, siehe Kapitel 19). Ist der Garbage-Collector aktiv, ist es ausreichend, den Listenkopf zu initialisieren. Existieren auf die Listenelemente keine sonstigen Referenzen, so entsteht hier Garbage, der vom Collector eingesammelt werden kann. Ohne Garbage-Collector muß jedes Listenelement einzeln freigegeben werden. Dies dauert nicht nur deutlich länger als das Initialisieren der Liste, hier besteht zudem die Gefahr, daß hängende Referenzen existieren, wenn andere Zeigervariablen auf die Listenelemente existieren.

```
PROCEDURE Clear(list: Lists.List);
VAR
  node: Lists.NodePtr;
BEGIN
   (* $IF GarbageCollector *)
     Lists.Init(list);
   (* $ELSE *)
     WHILE ~ list.isEmpty() DO
       node := Lists.RemHead(list);
       DISPOSE(node);
     END;
   (* $END *)
END Clear;
```

Das Programm GCStat

Dieses Programm zeigt quantitativ und qualitativ die Aktivität des Garbage-Collectors, und die Menge des über den Garbage-Collector verwalteten Speichers an. Nach dem Start durch einen Doppelklick auf sein Piktogramm oder Eingabe von 'GCStat' in eine Shell öffnet das Programm dieses Fenster:



In der Zeile direkt unter dem Fenstertitel wird die Menge des insgesamt vom System allozierten Speichers, die Anzahl der derzeit lebendigen und toten Objekte und die Nummer des aktuellen Collectorzyklus angezeigt. Lebendige Objekte (*Alive*) sind diejenigen Objekte, die der Garbage-Collector für erreichbar und von den Mutatoren benötigt hält. Diese Objekte können jedoch jederzeit zu Garbage werden, was der Collector schlechtestenfalls erst im nächsten Zyklus erkennt. Die toten Objekte (*Dead*) sind Objekte, die der Collector als unerreichbar erkannt hat, die jedoch so klein sind, daß er sie nicht an das System zurückgibt, sondern selbst zwischenspeichert, damit sie schneller wieder angefordert werden können.

Die Nummer des Collectorzyklus kann zur Bestimmung der Dauer eines Zyklus bestimmt werden.

Im großen Bereich des Fensters werden die zeitlichen Verläufe der Werte dargestellt: Der schwarze Bereich repräsentiert die lebendigen Objekte. Ist er hoch, so gibt es viele lebendige Objekte.

Der blaue Bereich (je nach der Farben-Voreinstellung können die Farben natürlich variieren) entspricht der Anzahl an toten Objekten. Zusammen mit dem schwarzen Bereich ergibt er die Gesamtzahl an Speicherbereichen, die der Garbage-Collector vom System angefordert hat.

Eine dünne weiße waagerechte Linie zeigt die Menge dieses Speichers in Bytes an.

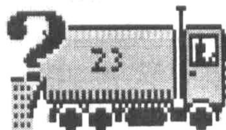
Kopierrecht der 'garbagecollector.library'

Die 'garbagecollector.library' darf zusammen mit Oberon Programmen, die mit Amiga Oberon übersetzt wurden, vertrieben oder verteilt werden. Dies gilt auch für die Kurzanleitung 'GarbageCollector.LiesMich' (auf der ersten Diskette von Amiga Oberon enthalten) und den Voreinstellungs-Editor GarbagePrefs, der im nächsten Kapitel beschrieben wird. Dabei ist eine Veröffentlichung als frei

kopierbare Software (etwa auf Public-Domain Diskettenserien wie z.B. [AMOK]) gleichermaßen erlaubt wie der Vertrieb als Teil eines kommerziellen Produktes.

Eine spezielle Erlaubnis für den Vertrieb der 'garbagecollector.library' ist lediglich nötig, wenn Programme Teil des Produktes sind, die diese Library benutzen, selbst jedoch nicht mit Amiga Oberon geschrieben wurden. In einem solchen Fall setzen Sie sich bitte mit dem Autor in Verbindung.

17. Der Voreinsteller GarbagePrefs



Im vorigen Kapitel wurde der Garbage-Collector beschrieben. Damit er auf verschiedenen Computern, mit zum Teil völlig unterschiedlichen Ausstattungen an Speicher und Rechenleistung, und zusammen mit den unterschiedlichen Arten von Programmen immer unter besten Bedingungen arbeiten kann, können verschiedene Einstellungen mit dem Programm GarbagePrefs verändert werden.

GarbagePrefs ist ein Voreinsteller ähnlich den von der Workbench bekannten Editoren für die Farbeinstellung, den Bildschirmmodus usw. Er sollte daher in das Verzeichnis 'Prefs' der Systempartition kopiert werden.

Aufruf von GarbagePrefs

Aufgerufen wird GarbagePrefs durch einen Doppelklick auf sein Piktogramm oder durch Eingabe von 'GarbagePrefs' in ein Shell-Fenster. Das Programm öffnet ein Fenster, mit dem verschiedene Werte verändert werden können:

Garbage-Collector Voreinstellungen

Collector-Task Priorität: [-5]   Auch Chip-RAM allozieren [X]

Auf Collector-Zyklus warten alle	2000	Objekte oder	2000000	Bytes
Maximal allozieren:	99999	Objekte oder	99999999	Bytes
Speicher freihalten: mindestens			100000	Bytes
Größten Block freihalten: mindestens			50000	Bytes
Freigehaltenen Speicher prüfen alle	10	Objekte oder	50000	Bytes
Größten Block prüfen alle	20	Objekte oder	50000	Bytes
Maximale Größe der gemerkten Objekte:			1024	Bytes
Maximal gemerkter Speicher:	1000	Objekte oder	1000000	Bytes

Speichern   Benutzen   Abbrechen

Die Eingabesymbole

Die Texteingabefelder und Symbole haben im Einzelnen die folgenden Bedeutungen:

Collector-Task Priorität

Die Priorität des Collectors, also des Hintergrundprozesses, der den nicht mehr verwendeten Speicher auffindet und freigibt, kann hier eingestellt werden. Die Priorität sollte auf jeden Fall kleiner als null sein, da sonst die gesamte Rechenzeit vom Collector verwendet wird, und mit anderen Programmen nicht mehr gearbeitet werden kann.

Die eingestellt Priorität darf nicht größer als die eines der Programme sein, die den Collector benutzen, da ansonsten dieses Programm keine Rechenzeit mehr bekommt. Voreingestellt ist die Priorität -5, die in den meisten Fällen sinnvoll ist.

Auch Chip-RAM allozieren

Ist dieses Symbol nicht angewählt, so alloziert der Garbage-Collector nur Fast-RAM, also keinen Speicher der für die Bildschirmdarstellung usw. verwendet wird ([RKM: Libraries 92]). Auf Systemen mit deutlich mehr Fast-RAM als Chip-RAM ist es sinnvoll kein Chip-RAM für gewöhnliche Objekte zu allozieren, da mit dem Fast-RAM sehr viel schneller gearbeitet werden kann. Zudem ist man dann relativ sicher, daß der Garbage-Collector nicht den gesamten Systemspeicher alloziert und somit ein Arbeiten mit dem Rechner unmöglich macht.

Auf Collector-Zyklus warten

Wie im vorigen Kapitel beschrieben, werden Programme, die den Garbage-Collector verwenden, und dem Collector zu wenig

Rechenzeit lassen und viele Objekte allozieren nach einer Bestimmten allozierten Menge angehalten, bis der Collector einen Zyklus beendet hat. Hier kann nun die Anzahl an Objekten und die maximale Speichermenge angezeigt, die angefordert werden darf, während der Collector einen Zyklus bearbeitet.

Werden hier zu kleine Werte eingetragen, so warten die Programme zu oft grundlos auf den Collector. Zu große Werte dagegen haben zur Folge, daß der Collector zu wenig Rechenzeit bekommt. Dann wird der Systemspeicher nach einiger Zeit knapp und das Programm muß für einen möglicherweise sehr aufwendigeren Collector-Zyklus länger angehalten werden.

Für die beiden Werte kann *-1* angegeben werden. Dann wird das Programm nicht angehalten, solange der Systemspeicher nicht ausgeht, auch wenn der Collector keine Rechenzeit bekommt und viele Objekte angefordert werden.

Maximal allozieren

Die hier eingetragenen Werte geben die Höchstzahl an Objekten und die maximale Speichermenge an, die der Garbage-Collector allozieren darf. Hat der Garbage-Collector diese Menge angefordert, so können von ihm keine weiteren Objekte angefordert werden.

Die eingetragenen Werte dienen zum 'Eindämmen' des Garbage-Collectors und zur Limitierung des Speichers, die die Programme anfordern, die ihn benutzen. Soll diese Speichermenge nicht beschränkt werden, reicht hier jeweils die Angabe *-1*.

Mit diesen Werten lassen sich Programme auch leicht unter (künstlichen) Speichermangelsituationen testen.

Speicher freihalten

Die angegebene Speichermenge wird vom Garbage-Collector immer freigehalten und bleibt dem System zur Verfügung. Damit man auch bei Speichermangel mit dem Rechner noch arbeiten kann, sollte man hier mindesten *20000* Bytes angeben. Alloziert der Garbage-Collector kein ChipRAM (s.o.), kann hier ruhig *0* angegeben werden.

Größten Block freihalten

Wie bei 'Speicher freihalten' wird hier die minimale Größe des größten freien Speicherblocks angegeben, den der Garbage-Collector für das System freihalten soll. Hier ist ein Wert von mindestens *10000* Bytes zu empfehlen. Wird kein Chip-RAM alloziert, kann auch hier *0* angegeben werden.

Freigehaltenen Speicher prüfen

Hier wird angegeben wie oft, also nach wie vielen Speicheranforderungen und nach welcher angeforderten Speichermenge, der bei *Speicher freihalten* angegebene Wert geprüft werden soll. Wird dies zu oft geprüft, so wird das Anfordern von Speicher deutlich langsamer. Wird jedoch zu selten geprüft, kann nicht garantiert werden, daß die angegebene Speichermenge auch freigehalten wird.

-1 gibt hier an, daß der freizuhaltende Speicher nicht geprüft werden soll. Dies sollte angegeben werden, wenn der Garbage-Collector kein Chip-RAM alloziert.

Größten Block prüfen

Hier wird angegeben wie oft, also nach wie vielen Speicheranforderungen und nach welcher angeforderten Speichermenge, der bei *Größten Block freihalten* angegebene Wert geprüft wer-

den soll. Wird dies zu oft geprüft, so wird das Anfordern von Speicher deutlich langsamer. Wird jedoch zu selten geprüft, kann nicht garantiert werden, daß der größte freie Speicherblock die angegebene Größe nicht unterschreitet.

-1 gibt hier an, daß der größte Block nicht geprüft werden soll. Dies sollte angegeben werden, wenn der Garbage-Collector kein Chip-RAM alloziert.

Maximale Größe gemerkter Objekte

Der hier angegebene Wert bestimmt die maximale Größe der kleinen Objekte, die der Garbage-Collector nicht an das System zurückgeben soll, sondern selbst zwischenspeichert (siehe voriges Kapitel). Der angegebene Werte sollte wenige tausend Bytes nicht übersteigen.

Maximal gemerkter Speicher

Die angegebenen Werte bestimmen die maximale Anzahl und die Höchstmenge an Speicher, den die vom Garbage-Collector zwischengespeicherten kleinen Objekte belegen dürfen.

Dieser Speicher wird automatisch an das System zurückgegeben, wenn eine Speicheranforderung eines Programms fehlschlägt. Auf diese Weise geht durch das Zwischenspeichern praktisch kein Speicher verloren.

Wird sehr viel Speicher zwischengespeichert, kann das automatische Freigeben unter Umständen sehr lange dauern. Da der Speicher bei abgeschaltetem Multitasking freigegeben werden muß, kann dies einen kurzen Stillstand des Computers zur Folge haben. Wird bei *Maximal gemerkter Speicher* nur ein verhältnismäßig kleiner Wert, also etwa *100 000* Bytes angegeben, so bemerkt man diesen kurzen Stillstand praktisch nicht.

Speichern, Benutzen, Abbrechen

Diese Symbole bewirken dasselbe wie die entsprechenden Symbole der anderen Voreinsteller. *Speichern* übernimmt die Änderungen und sichert sie so, daß sie beim Neustart des Computers weiterhin aktiv bleiben. *Benutzen* aktiviert die Änderungen, bei einem Neustart werden sie jedoch wieder gelöscht. *Abbrechen* beendet das Programm, ohne die Voreinstellungen zu aktivieren.

Damit die neuen Voreinstellungen gleich aktiviert werden können, darf kein Programm die Garbage-Collector-Library verwenden. Ansonsten werden die Änderungen erst dann aktiv, wenn alle Programme beendet wurden, die die Library verwenden.

Die Menüs

Projekt

Mit den Punkten dieses Menüs können die Voreinstellungen in einer Datei gespeichert (*Speichern als...*) und aus einer Datei geladen (*Öffnen...*) werden.

Der Menüpunkt *Beenden* hat die gleiche Funktion wie das Symbol *Abbrechen.*

Editieren

Mit diesem Menü können die Werte auf die *voreingestellten Werte* oder die *zuletzt gespeicherten Werte* gesetzt werden. *Änderungen zurücknehmen* setzt die Werte wieder so, wie sie beim Start von GarbagePrefs waren.

Der wichtigste Punkt dieses Menüs ist *Werte vorschlagen.* Wird er angewählt, so untersucht GarbagePrefs den Rechner auf dem es ge-

startet wurde. Es berücksichtigt dabei die Menge an Chip-RAM, die Größe des Fast-RAMs und den derzeit freien Speicher. Aus diesen Informationen berechnet GarbagePrefs Werte für die verschiedenen Einstellungen, die im Normalfall ein sinnvolles Arbeiten der Garbage-Collector-Library ermöglichen.

Wurden die Voreinstellungen nicht mit GarbagePrefs eingestellt, so bestimmt die Library auf diese Weise die Voreinstellungen, die sie verwendet. So kann die Library also gewöhnlich auch dann verwendet werden, wenn lediglich die Datei 'garbagecollector.library' in das Verzeichnis 'LIBS:' kopiert wurde.

Settings

Dieses Modul bietet ein mit einem Häkchen versehenen Menüpunkt *Piktogramme erzeugen?*. Ist er angewählt, so werden für die mit *Speichern als...* des *Projekt*-Menüs gespeicherten Dateien Piktogramme erzeugt.

18. Der erzeugte Code



Dieses Kapitel beschreibt den vom Compiler erzeugten Programmcode. Für das Verständnis dieses Kapitels sind Assemblerkenntnisse erforderlich, eine gute Referenz für die Assemblerprogrammierung der MC68000-Prozessorfamilie ist in [Williams 89] enthalten. Da der genaue Aufbau des erzeugten Codes für reine Oberon-Programmierer irrelevant ist, können sie dieses Kapitel überspringen.

Aufbau der Oberon-Programme

Jedes mit Amiga Oberon compilierte Modul (mit der Ausnahme *OberonLib*) läßt sich zu einem ausführbaren Programm linken. Um dies zu ermöglichen, enthält der erste Hunk jeder Objektdatei (zum Aufbau der Objektdateien siehe [AmigaDOS 91] den zum Start des Programms nötigen Code. Dieser Code teilt OberonLib über die Variable *OberonLib.HaltProc* die Adresse des CLOSE-Teils des Hauptmoduls mit und speichert das Register A7 in *OberonLib.OldSP*.

Beim kleinen Datenmodell alloziert der Startcode Speicher für die globalen Variablen. Gibt es nicht genügend Speicher, wird das Proramm sofort beendet, bei einem Workbench-Start wird noch die Workbench-Startup-Message ([RKM: Libraries 92]) beantwortet.

Nun ruft der Startcode den BEGIN-Anweisungsteil des Hauptmoduls auf. Nach der Rückkehr davon, was durch ein *RTS* oder einen Aufruf von *HALT* geschieht, wird der Inhalt von A7 wieder auf *OberonLib.OldSP*, *OberonLib.closing* auf *TRUE* gesetzt und der CLOSE-Anweisungsteil des Hauptmoduls aufgerufen. Zum Schluß wird beim kleinen Datenmodell noch der Speicher der globalen Variablen freigegeben und mit *RTS* das Programm beendet.

Aufruf der BEGIN- und CLOSE-Anweisungen

Jedes Modul enthält eine versteckte globale *INTEGER*-Variable, die beim Aufruf des BEGIN- und des CLOSE-Anweisungsteils verwendet wird. Diese Variable wird im folgenden als *OpenCnt* bezeichnet. Die Objektdatei jedes Moduls enthält diese Anweisungsteile, sie sind jedoch nur eventuell leer.

Die obersten 15 Bit von *OpenCnt* enthalten einen Zähler, der angibt, wie oft der BEGIN-Anweisungsteil aufgerufen wurde. Wird ein Modul von mehreren anderen Modulen importiert, so wird der BEGIN-Anweisungsteil möglicherweise mehrmals aufgerufen. Er wird jedoch nur beim ersten Aufruf ausgeführt. Bei der Ausführung des BEGIN-Anweisungsteil wird zunächst der Zähler in den obersten 15 Bit von *OpenCnt* um eins erhöht. Danach werden die BEGIN-Anweisungen aller von diesem Modul importieren Module aufgerufen. Nun wird das unterste Bit des *OpenCnt* gesetzt. Dadurch wird angezeigt, daß der BEGIN-Teil wirklich ausgeführt wird, und das Programm nicht beim Ausführen der BEGIN-Anweisungen eines importierten Moduls abgebrochen wurde.

Beim Ausführen der CLOSE-Anweisungen geschieht dies nun rückwärts: Der Zähler in den oberen 15 Bits wird jedesmal um eins verringert. Erreicht er null wird geprüft, ob das unterste Bit gesetzt ist, also ob der BEGIN-Anweisungsteil dieses Moduls ausgeführt wurde. Dann wird auch der CLOSE-Anweisungsteil ausgeführt. Zuletzt werden noch die CLOSE-Anweisungen der importierten Module ausgeführt.

Zugriff auf strukturierte Konstanten

Auf strukturierte Konstanten, speziell auf konstante Zeichenketten, wird grundsätzlich mit der absoluten Adressierung zugegriffen. Konstanten sind so 'selten', daß es sich nicht lohnen würde, um für sie ein Adressregister zu reservieren und auf sie relativ dazu zuzugreifen.

Jede strukturierte Konstante wird in einem eigenen Hunk gespeichert, so daß unbenutzte Konstanten beim Linken nicht in das Programm eingebunden werden. Dies ist z.B. dann zu beachten, wenn man einen Versionsstring (wie etwa *"\o$VER: Oberon 5.0 23-Aug-94"*) in ein Programm einbinden möchte. Er kommt nur dann in das ausführbare Programm, wenn er einer Variablen zugewiesen wird.

Zugriff auf globale Variablen

Auf globale Variablen wird gewöhnlich relativ zum Adreßregister A5 zugegriffen. Bei Verwendung des großen Datenmodells wird auf die Variablen, deren Adresse von der Basisadresse der Variablen mindestens 32768 Bytes entfernt liegt, mit der absoluten Adressierung zugegriffen. Beim großen Datenmodell hat jedes Modul einen eigenen Bereich für die globalen Variablen. Daher wird hier in jeder globalen Prozedur A5 zunächst auf die Adresse dieses Bereichs gesetzt.

Im kleinen Datenmodell ist A5 über den gesamten Programmlauf konstant. Auf alle Variablen wird relativ zu A5 zugegriffen, daher darf der Variablenbereich eine Größe von 32768 Bytes auch nicht übersteigen.

Prozeduren

Globale Prozeduren werden zusammen mit den zu ihnen lokalen Prozeduren in jeweils einem Hunk gespeichert. Dadurch werden beim Linken nur die verwendeten Prozeduren in das ausführbare Programm eingebunden, es wird also optimierend gelinkt.

Eine globale Prozedur lädt beim großen Datenmodell A5 mit dem Zeiger auf den Bereich der globalen Variablen des Moduls. Danach erniedrig sie das Register A7, damit auf dem Stapelspeicher oberhalb von A7 genügend Speicher für die lokalen Variablen der Prozedur ist.

Bei Prozeduren mit offenen Feldern als Wertparametern werden diese

jetzt noch auf den Stapel kopiert. Da hier der verwendete Stapelspeicher keine feste Größe hat, wird die Anfangsadresse der lokalen Variablen nach A4 geladen und auf die Variablen wird relativ zu A4 zugegriffen. Bei anderen Prozeduren ist dies nicht nötig, dort wird auf die lokalen Variablen direkt relativ zu A7 zugegriffen.

Am Schluß jeder Prozedur wird A7 wieder erhöht, und die Rücksprungadresse wird vom Stapelspeicher geholt. Nun gibt die Prozedur den Speicher der Prozedurparameter frei und springt an die Rücksprungadresse.

Aufbau des Stapelspeichers bei globalen Prozeduren

Der Stapelspeicher einer globalen Prozedur enthält die folgenden Daten:

	von früher aufgerufenen Prozeduren belegt
	Prozedurparameter
	Rücksprungadresse
	gerettetes A5 (großes Datenmodell)
	lokale Variablen
A7/A4 →	frei

A7 bzw. A4 (bei Prozeduren mit offenen Feldern als Wertparametern) zeigt dabei auf die unterste Adresse der lokalen Variablen. Wird relativ zu A7 auf die lokalen Variablen zugegriffen, so bestimmt der Compiler die jeweils gültigen Offsets relativ zu A7, da sich A7 während der Ausführung der Prozedur ändern kann, etwa durch den Aufruf einer anderen Prozedur und das Speichern der Parameter auf den Stapel.

Aufbau des Stapelspeichers bei lokalen Prozeduren

Der Stapelspeicher einer lokalen Prozedur enthält die folgenden Daten:

	von früher aufgerufenen Prozeduren belegt
	Prozedurparameter
	Zeiger auf Variablen der äußeren Prozedur
	Rücksprungadresse
	lokale Variablen
A7/A4 →	frei

Über den Zeiger auf die lokalen Variablen der umschließenden Prozedur kann diese Prozedur auf die lokalen Variablen aller umschließenden Prozeduren zugreifen. Bei mehrfach geschachtelten Prozeduren muß über diesen Zeiger aus den lokalen Variablenbereich der jeweils nächste Zeiger auf die lokalen Variablen geladen werden. Bei den Prozeduren

```
PROCEDURE A;
VAR
   i: INTEGER;
   PROCEDURE B;
     PROCEDURE C;
     BEGIN
       i := 3;
     END C;
   ...
   END B;
...
END A;
```

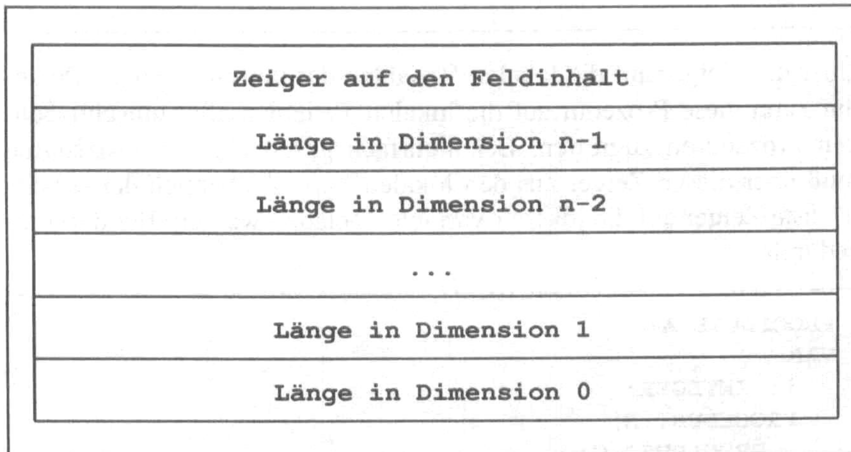
muß für die Zuweisung i := 3 der folgende Code erzeugt werden:

```
0000: 286F0004          MOVEA.L  4(A7),A4
0004: 286C0004          MOVEA.L  4(A4),A4
0008: 38BC0003          MOVE.W   #$0003,(A4)
```

Es wird zuerst der Zeiger auf die lokalen Variablen von B nach A4 geladen. Aus den lokalen Variablen von B wird der Zeiger auf die lokalen Variablen von A entnommen. Mit ihm kann letztlich auf i zugegriffen werden.

Aufbau der offenen Feldparameter

Bei der Parameterübergabe von offenen Feldparametern wird von der Seite des Aufrufers kein Unterschied zwischen VAR- und Wertparametern gemacht. Sie sind wie folgt aufgebaut:



Die Dimension des Feldes ist n. Die Längen in den verschiedenen Dimensionen sind jeweils 32-Bit-Werte. Bei VAR-Parametern wird der Inhalt des Feldes von der Prozedur auf den Stapelspeicher kopiert und der Zeiger auf den Feldinhalt wird entsprechend auf das kopierte Feld geändert.

Funktionsergebnisse

Gewöhnliche Funktionsergebnisse, die in ein 32-Bit Register passen, werden, wie es auf dem Amiga üblich ist, im Prozessorregister D0 zurückgegeben. *LONGREAL*-Zahlen werden über die beiden Registern D0 und D1 verteilt zurückgegeben.

Eine Rückgabe über Prozessorregister ist mit verfolgten Zeigern jedoch nicht möglich, da ein verfolgter Zeiger immer auch in einer vom Garbage-Collector erreichbaren verfolgten Variablen stehen muß, was für ein Ergebnis in einem Register nicht garantiert werden kann. Auch Ergebnistypen, die nicht in einem Register aufgenommen werden können, wie Strukturen oder Zeiger auf offene Felder (s.u.), muß ein anderer Weg gewählt werden.

Amiga Oberon behandelt solche Ergebnisse ähnlich wie VAR-Parameter. Der Aufrufer übergibt dabei vor der eigentlichen Parameterliste einen versteckten VAR-Parameter. In diesen Parameter schreibt die Prozedur ihr Resultat. Der Compiler erzeugt lokal zu der Prozedur oder dem Module, in dem der Aufruf vorkommt, eine versteckte Variable für das Resultat.

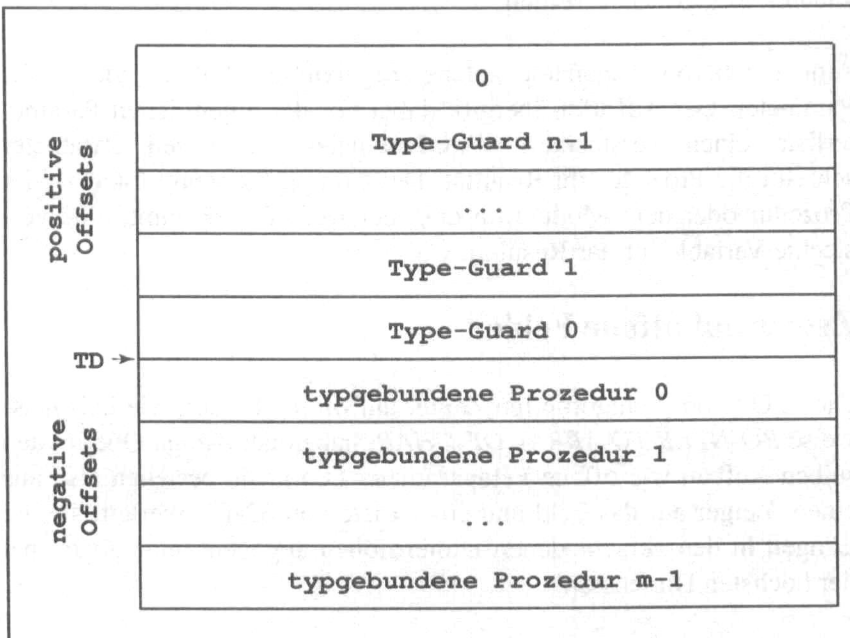
Zeiger auf offene Felder

Die in Oberon-2 eingeführten Zeiger auf offene Felder, wie beispielsweise *POINTER TO ARRAY OF CHAR*, haben bei Amiga Oberon den selben Aufbau wie offene Feldparameter (s.o.). Sie bestehen also aus einem Zeiger auf das Feld und einer Liste von 32-Bit Werten, die die Längen in den verschiedenen Dimensionen angeben, angefangen bei der höchsten Dimension.

Dies ist auch der Grund, weshalb diese Zeiger nicht zuweisungskompatibel zum Typ *ADDRESS* aus *SYSTEM* sein können (Kapitel 14).

Aufbau der Typdescriptoren

Jede Variable eines Recordtyps enthält als erstes (verstecktes) Element einen Zeiger auf eine Struktur, die ihren Typ beschreibt, den sogenannten Typdescriptor. Diese Struktur enthält zum einen die Information, um welchen Typ es sich handelt, also ob und wenn ja welche Erweiterung des Grundtyps dieser Recordtyp ist. Zum anderen enthält der Typdescriptor die Adressen der typgebundenen Prozeduren des Typs. Bei einem Aufruf einer typgebundenen Prozedur wird nicht an eine statische Adresse sondern an die im Typdescriptor angegebene Adresse gesprungen, so daß derselbe Aufruf bei unterschiedlichen Typen des Zielobjekts in unterschiedliche Prozeduren springt (dynamisches Binden).



Dieser Typdescriptor beschreibt einen Recordtyp mit n Erweiterungsstufen (sein Basistyp wurde n-1 mal erweitert) und m typgebundenen Prozeduren.

Der Zeiger auf den Typdescriptor (TD), der in die Recordvariablen eingetragen wird, zeigt auf den ersten Type-Guard, hier also Type-Guard 0. Mit positiven Offsets kann auf die Type-Guards, mit negativen Offsets auf die typgebundenen Prozeduren zugegriffen werden.

Die Type-Guards werden beim Typ-Test (mit IS) und bei der Typüberprüfung (bei *WITH* oder bei Angabe eine Typeguards in runden Klammern) benötigt. Sie sind für jeden Recordtyp eindeutige 32-Bit-Werte. Der Type-Guard mit der Nummer 0 ist der des Basistyps, die mit der Nummer k sind die der k-ten Erweiterung.

Die Adressen der typgebundenen Prozeduren werden für das dynamische Binden beim Aufruf von typgebundenen Prozeduren benötigt. Sie enthalten die Adressen der entsprechenden Prozeduren. Jeder typgebundenen Prozedur teilt der Compiler einen festen Offset zu. Neu definierte typgebundene Prozeduren bekommen jeweils einen neuen Offset. Redefinierte typgebundene Prozeduren behalten den Offset der ursprünglichen Prozedur, so daß sie diese ersetzen.

Erzeugter Code bei Verwendung des Garbage-Collectors

Soll ein Programm seinen Speicher vom Garbage-Collector verwalten lassen, so müssen die Zeigervariablen des Programms speziell gehandhabt werden. Dem Garbage-Collector muß die Existenz jeder globalen und lokalen Zeigervariablen mitgeteilt werden. Außerdem muß bei der Zuweisung einer Zeigervariablen an eine andere das Objekt, auf das die zugewiesene Variable zeigt, für den Collector markiert werden, da der Collector dieses Objekt sonst falscherweise für unerreichbar halten könnte.

Die folgende Prozedur hat lokale Variablen, die verfolgte Zeiger enthalten:

```
PROCEDURE Traced;

VAR
  a: Lists.NodePtr;
  b: Lists.List;

BEGIN
  ...
END Traced;
```

Um dem Garbage-Collector zu Beginn der Prozedur mitzuteilen, daß es nun weitere verfolgte Variablen gibt, wird der folgende Code erzeugt:

```
0008: 2F7C00000000001C MOVE.L   #$00000000,28(A7)
0010: 28780004         MOVEA.L  $0004,A4
0014: 266C0114         MOVEA.L  276(A4),A3
0018: 286B002E         MOVEA.L  46(A3),A4
001C: 2654             MOVEA.L  (A4),A3
001E: 2F6B000C0018     MOVE.L   12(A3),24(A7)
0024: 49EF0018         LEA      24(A7),A4
0028: 274C000C         MOVE.L   A4,12(A3)
HUNK_EXT:
32 bit references on: Test_GCVARS_00000000
00000000: 0000000A ....
```

Zunächst wird hier ein Zeiger auf eine Struktur, die die lokalen Variablen beschreibt, in den lokalen Variablen gespeichert (bei Adresse *0008*). Nun wird der Bereich der lokalen Variablen in eine Liste der Mutator-Struktur (siehe Modul *GarbageCollector*, Kapitel 26) dieses Prozesses eingetragen. Auf diese Struktur kann über *execbase.thisTask.trapData* zugegriffen werden (genaueres dazu im Modul *OberonLib*, Kapitel 26). Die Liste der lokalen Variablen ist eine einfache vorwärts verkettete Liste, so daß die Variablen schnell als neues erstes Element eingetragen werden können.

Am Schluß der Prozedur muß dem Garbage-Collector noch mitgeteilt

werden, daß die lokalen Variablen jetzt nicht mehr existieren. Dies geschieht durch

```
0034: 28780004          MOVEA.L  $0004,A4
0038: 266C0114          MOVEA.L  276(A4),A3
003C: 286B0058          MOVEA.L  88(A3),A4
0040: 2654              MOVEA.L  (A4),A3
0042: 47EB000C          LEA      12(A3),A3
0046: 2853              MOVEA.L  (A3),A4
0048: 2694              MOVE.L   (A4),(A3)
```

Wie zuvor wird hier der Mutator über *execbase.thisTask.trapData* ermittelt. Dann wird schlicht das erste Element der Liste der lokalen Variablen entfernt

Die zweite wichtige Operation, für die spezieller Code für den Garbage-Collector erzeugt werden muß, ist die Zuweisung. So wird für die Zuweisung *a := b* in dem Programm

```
MODULE Test;

VAR
  a,b: POINTER TO RECORD END;

BEGIN
  a := b;
END Test.
```

der folgende Code erzeugt:

```
0058: 286D0000          MOVEA.L  0(A5),A4
005C: 2E0C              MOVE.L   A4,D7
005E: 670A              BEQ      $0000006A
0060: 546CFFFA          ADDQ.W   #2,-6(A4)
0064: 08EC0000FFFB      BSET     #$0000,-5(A4)
006A: 2B4C0004          MOVE.L   A4,4(A5)
006E: 6704              BEQ      $00000074
0070: 556CFFFA          SUBQ.W   #2,-6(A4)
```

Durch die Anweisungen *ADDQ.W #2,-6(A4)* und *SUBQ.W #2,-6(A4)* wird das Objekt *b^* vor Veränderungen durch den Collector kurzzeitig geschützt. Während es geschützt ist, wird es durch *BSET #$0000,-5(A4)* markiert und mit *MOVE.L A4,4(A5)* an *a* zugewiesen.

Dieser recht aufwenige Code, der für den Garbage-Collector nötig wird, ist die Hauptursache für den deutlich längeren Code im Vergleich zur Übersetzung ohne Garbage-Collector. Dennoch arbeiten Programme mit dem Garbage-Collector erstaunlich effizient und es ist meist nur schwer ein Unterschied zur Übersetzung ohne Garbage-Collector zu bemerken.

Der Überprüfungscode

Für die in Kapitel 14 beschriebenen Arten von Kontrollen, die über Optionen aktiviert werden können, muß zusätzlicher Programmcode erzeugt werden. Dieser wird im folgenden beschrieben:

Stackkontrolle

Bei der Stackkontrolle wird am Anfang jeder Anweisungsfolge, die durch *BEGIN* oder *CLOSE* eingeleitet wird, Code erzeugt werden, der prüft, ob der Stapelspeicher für die lokalen Variablen und Prozeduraufrufe dieser Anweisungsfolge ausreicht. Dazu wird die Prozedur *StackChk* aus *OberonLib* (Kapitel 26) verwendet. Sie erhält in D0 die Größe des benötigten Stapelspeichers. Zusätzlich sorgt diese Prozedur für einen Programmabbruch bei Steuerung und 'C', wenn eines der Module *Break* und *BreakRq* (Kapitel 25) verwendet wird.

Eine leere globale Prozedur mit 12 Bytes an lokalen Variablen wird folgendermaßen übersetzt:

```
0000: 700C              MOVEQ    #12,D0
0002: 4EB900000000      JSR      $00000000
0008: 42A7              CLR.L    -(A7)
000A: 42A7              CLR.L    -(A7)
000C: 42A7              CLR.L    -(A7)
000E: 4FEF000C          LEA      12(A7),A7
0012: 4E75              RTS
HUNK_EXT:
relocatable definition: Test_AC = 00000000 (0)
32 bit references on: OberonLib.StackChk
00000000: 00000004 ....
```

Überlaufskontrolle

Ein Überlauf wird anhand des Overflow-Flags des MC68000 erkannt. Dies geschieht mit Hilfe des Befehls *TRAPV*.

Die Anweisung *INC(i)*, die eine *INTEGER*-Variable um eins erhöht, wird mit Überlaufskontrolle folgendermaßen übersetzt:

```
002A: 526D0002          ADDQ.W   #1,2(A5)
002E: 4E76              TRAPV
```

Bereichskontrolle

Die Bereichskontrolle prüft beim Zugriff auf Feld- und Mengenelemente und beim Umwandeln von Integerzahlen mit *SHORT*, ob der zulässige Wertebereich eingehalten wird.

So wird für die Anweisung *c := s[i]* in dem folgenden Programmstück

```
MODULE RangeChk;

VAR
   s: ARRAY 80 OF CHAR;
   i: INTEGER;
   c: CHAR;

BEGIN
   c := s[i];
END RangeChk.
```

Der folgende Code erzeugt:

```
002E: 3E2D0050       MOVE.W  80(A5),D7
0032: 4FBC004F       CHK.W   #$004F,D7
0036: 1B7570000052   MOVE.B  0(A5,D7.W),82(A5)
```

Case-Index-Kontrolle

Diese Überprüfung ist sehr einfach: Es wird für den fehlenden ELSE-Teil eine CASE-Anweisung ein *TRAP #1* eingefügt. Da diese Anweisung in einem korrekten Programm nie angesprungen wird, wird dadurch die Programmausführung auch nicht verlangsamt.

Return-Kontrolle

Auch diese Überprüfung ist mit wenig Aufwand verbunden: Es wird als letzter Befehl einer Funktionsprozedur ein *TRAP #4* eingefügt. Fehlt eine RETURN-Anweisung, so wird das Programm durch diesen Befehl abgebrochen.

NIL-Zeiger-Kontrolle

Die Überprüfung, ob verwendete Zeigervariablen den Wert *NIL* enthalten, ist etwas aufwendiger. Der Zeiger wird hier zunächst

in ein Adreßregister geladen, daß dann in ein Datenregister kopiert wird, um die Statusflags korrekt zu setzen. Nun kann bei gesetzten Zero-Flag mit einem *TRAP #3* abgebrochen werden. So wird in dem Programm

```
MODULE NilChk;

VAR
   p : POINTER TO RECORD a,b,c: INTEGER END;
   i : INTEGER;

BEGIN
   i := p.b;
END NilChk.
```

für die Zuweisung *i := p.b* der folgende Code erzeugt:

```
0054: 286D0000         MOVEA.L 0(A5),A4
0058: 2E0C             MOVE.L  A4,D7
005A: 6602             BNE     $0000005E
005C: 4E43             TRAP    #3
005E: 3B6C00060004     MOVE.W  6(A4),4(A5)
```

Ungerade-Zeiger-Kontrolle

Diese Kontrolle ist nur auf einem MC68020 oder einem neueren Prozessor sinnvoll. Für die Zuweisung *i := p.b* im oberen Programm wird zusammen mit der NIL-Zeiger-Kontrolle der folgende Code erzeugt:

```
0054: 286D0000         MOVEA.L 0(A5),A4
0058: 2E0C             MOVE.L  A4,D7
005A: 6602             BNE     $0000005E
005C: 4E43             TRAP    #3
005E: E29F             ROR.L   #$01,D7
0060: 6A02             BPL     $00000064
0062: 4E47             TRAP    #7
0064: 3B6C00060004     MOVE.W  6(A4),4(A5)
```

Typkontrolle

Die Typkontrolle wird beim Zugriff auf Records mit der WITH-Anweisung und bei der Angabe eines Typeguards nötig. Dabei wird auf den Typdescriptor des Records zugegriffen. Es werden alle Erweiterungsstufen, die zwischen dem statischen Typ der Recordvariable und dem zu prüfenden Typen liegen, nacheinander geprüft. Stimmen sie alle, so ist die Variable wirklich von dem angegebenen Typ.

In dem Programm

```
MODULE TypeChk;

TYPE
   A = RECORD      a,b: INTEGER END;
   B = RECORD (A) c,d: INTEGER END;
   C = RECORD (B) e,f: INTEGER END;

VAR
   p,q: POINTER TO A;

BEGIN
   p(C).e := q(B).d;
END TypeChk.
```

wird für die Anweisung *p(C).e := q(B).d* bei abgeschalteter NIL-Zeiger-Kontrolle der recht aufwendige folgende Code erzeugt:

```
0052: 49ED000C          LEA      12(A5),A4
0056: 274C0008          MOVE.L   A4,8(A3)
005A: 286D0004          MOVEA.L  4(A5),A4
005E: 2E0C              MOVE.L   A4,D7
0060: 6718              BEQ      $0000007A
0062: 2654              MOVEA.L  (A4),A3
0064: 0CAB24BCD801..    CMPI.L   #$24BCD801,4(A3)
006C: 660A              BNE      $00000078
```

```
006E: 0CAB24BCD802.. CMPI.L   #$24BCD802,8(A3)
0076: 6702           BEQ      $0000007A
0078: 4E45           TRAP     #5
007A: 266D0000       MOVEA.L  0(A5),A3
007E: 2E0B           MOVE.L   A3,D7
0080: 670E           BEQ      $00000090
0082: 2453           MOVEA.L  (A3),A2
0084: 0CAA24BCD801.. CMPI.L   #$24BCD801,4(A2)
008C: 6702           BEQ      $00000090
008E: 4E45           TRAP     #5
0090: 396B000A000C   MOVE.W   10(A3),12(A4)
```

Es wird zunächst geprüft, ob die Zeigervariable *p* nicht nur auf ein Record des Typs *A*, sondern sogar auf eines des zweifach erweiterten Typs *C* zeigt. Danach wird geprüft, ob *q* auf ein Record des Typs *B* zeigt, was mit einem einzigen Test geschehen kann.

Automatisch abgefange Fehler

Für Fehler wie eine Division durch null wird kein spezieller Überprüfungscode erzeugt, da ein solcher Fehler automatisch zu einer Ausnahmesituation führt. Daher kann das Abfangen dieser Fehler auch nicht abgeschaltet werden.

Listenparameter

Die an einen Listenparameter (Taglisten, der zweite Parameter von *Dos.PrintF*, usw.) übergebenen Werte werden in einer vom Compiler erzeugten Variablen gespeichert. Diese Variable wird lokal zum Sichtbarkeitsbereich des aktuellen Anweisungsteils gespeichert, also innerhalb einer Prozedur als lokale Variable dieser Prozedur und in einen BEGIN- oder CLOSE-Anweisungsteil als globale Variable des Moduls.

19. Modulbibliothek: Datenstrukturen



Die in diesem Kapitel beschriebenen Module stellen eine Reihe an Datenstrukturen für das Arbeiten mit Oberon zur Verfügung. Hier werden jeweils nur die mit ModToDef (Kapitel 8) erzeugten Definitionsmodule aufgelistet. Die kompletten Quelltexte der Module befinden sich auf den Amiga Oberon Disketten.

AVL

```
DEFINITION AVL;

IMPORT BT := BasicTypes;

TYPE
  NodePtr = POINTER TO Node;
  Node = RECORD (BT.ANYDesc)
    l : NodePtr;
    r : NodePtr;
  END;
  RootPtr = POINTER TO Root;
  Root = RECORD (BT.COLLECTIONDesc)
    root : NodePtr;
    PROCEDURE (tree:RootPtr) Add(x: BT.ANY);
    PROCEDURE (tree:RootPtr) Remove(x: BT.ANY);
    PROCEDURE (tree:RootPtr) nbElements(): LONGINT;
    PROCEDURE (tree:RootPtr) Do(p: BT.DoProc;
                                   par: BT.ANY);
  END;
  CompProc = PROCEDURE(a, b: NodePtr): INTEGER;
  FindProc = PROCEDURE(VAR root: Root;
                           a: NodePtr): INTEGER;
  DoProc = PROCEDURE(a: NodePtr);
```

```
CONST
  left = -1;
  ok = 0;
  right = 1;

TYPE
  String = ARRAY  OF CHAR;
  SNodePtr = POINTER TO SNode;
  SNode = RECORD (Node)
    name : String;
  END;
  SRoot = RECORD (Root) END;

PROCEDURE Init(VAR root: Root;
                   cmp: CompProc;
                   find: FindProc);
PROCEDURE Add(VAR root: Root;
                  node: NodePtr): BOOLEAN;
PROCEDURE Find(VAR root: Root): NodePtr;
PROCEDURE Remove(VAR root: Root;
                     node: NodePtr): BOOLEAN;
PROCEDURE DoForward(root: Root; proc: DoProc);
PROCEDURE DoBackward(root: Root; proc: DoProc);
PROCEDURE Dispose(VAR root: Root);
PROCEDURE SInit(VAR root: SRoot);
PROCEDURE SAdd(VAR root: SRoot;
                   node: SNodePtr): BOOLEAN;
PROCEDURE SFind(VAR root: SRoot;
                    str: String): SNodePtr;

END AVL.
```

Dieses Modul stellt Typen für Knoten und Wurzeln von AVL-Bäumen und Prozeduren zu deren Bearbeitung zur Verfügung. AVL-Bäume sind spezielle ausgeglichene binäre Bäume. Eine Beschreibung dieser Bäume findet man beispielsweise in [Wirth 83].

Um dieses Modul zu benutzen, definiert man sich eine Erweiterung des Typs *Node*, und fügt die zu speichernden Daten mit der Erweite-

rung an. Zudem müssen zwei Prozeduren *Comp* und *Find* definiert werden, die folgendermaßen aufgebaut sein müssen:

PROCEDURE Comp(a,b: NodePtr): INTEGER;
PROCEDURE Find(VAR root: Root; a: NodePtr): INTEGER;

Comp wird zum Vergleichen zweier Elemente benötigt. Das Ergebnis muß folgender Wert sein:

```
Comp(a,b) < 0, falls a^ < b^
Comp(a,b) = 0, falls a^ = b^
Comp(a,b) > 0, falls a^ > b^
```

Find wird beim Suchen nach einem Element verwendet. Hier muß gelten:

```
Find(root,a) < 0, falls a^ < gesuchtes Element
Find(root,a) = 0, falls a^ = gesuchtes Element
Find(root,a) > 0, falls a^ > gesuchtes Element
```

Der Parameter *root* kann zum Speichern der Eigenschaften des gesuchten Elementes verwendet werden. Dazu muß als Wurzel des Baumes eine Erweiterung des Typs *Root* definiert werden.

PROCEDURE Init(VAR root: Root;
comp: CompProc;
find: FindProc);

Diese Prozedur muß zum Initialisieren und Leeren des Baumes aufgerufen werden, bevor er von anderen Prozeduren verwendet wird. Übergeben wird die Wurzel und die beiden oben beschriebenen Prozeduren.

PROCEDURE Add(VAR root: Root;
node: NodePtr): BOOLEAN;

Add fügt das Element *node* in den Baum ein. Das Ergebnis ist *TRUE*, wenn das Einfügen erfolgreich war, also wenn noch kein gleiches Element in dem Baum existierte.

PROCEDURE Find(VAR root: Root): NodePtr;

Find sucht mit Hilfe der bei *Init* angegebenen *find*-Prozedur nach einem Element des Baumes. Wurde es gefunden, so liefert *Find* einen Zeiger auf dieses Element, ansonsten ist das Ergebnis *NIL.*

PROCEDURE Remove(VAR root: Root;
VAR node: NodePtr): BOOLEAN;

Das Element *node* wird aus dem Baum entfernt. Das Ergebnis ist *TRUE* wenn das Element im Baum enthalten war und entfernt werden konnte.

PROCEDURE DoForward(root: Root;
proc: DoProc);

Die Prozedur *proc* wird in aufsteigender Reihenfolge (in Infix-Reihenfolge) mit allen Elementen des Baumes als Parameter aufgerufen.

PROCEDURE DoBackward(root: Root;
proc: DoProc);

Wie bei *DoForward* wird *proc* mit allen Elementen des Baumes aufgerufen, hier jedoch in umgekehrter Reihenfolge, das letzte Element also zuerst und das erste zuletzt.

PROCEDURE Dispose(VAR root: Root);

Diese Routine löscht den gesamten Baum. Wird mit Garbage-Collector gearbeitet, setzt sie einfach die Wurzel des Baumes auf *NIL*. Wird dagegen ohne Garbage-Collector compiliert, so wird der Speicher jedes Elementes mit *DISPOSE* freigegeben.

Zeichenketten-AVL-Bäume:

In den meisten Fällen werden AVL-Bäume zum alphabetisch sortiertem Speichern von Elementen benötigt, deren Namen als Zeichenketten vorliegen. Dafür bietet das Modul *AVL* spezielle Typen und Prozeduren an: *SNode* ist ein Knoten der einen Namen enthält, *SRoot* wird für Wurzeln dieser Bäume benötigt, es enthält zusätzliche Informationen für das Suchen nach Knoten mit einem bestimmten Namen.

Bis auf *Init* und *Find* können auch mit diesen Typen alle oben beschriebenen Prozeduren verwendet werden. *SAdd* ist lediglich ein Synonym für *Add*. Statt *Init* muß *SInit* und statt *Find* muß *SFind* verwendet werden:

PROCEDURE SInit(VAR root: SRoot);

Hier werden keine weiteren Parameter benötigt, die Vergleichsprozeduren werden automatisch gesetzt.

PROCEDURE SFind(VAR root: SRoot;
str: String): SNodePtr;

Es wird nach dem Knoten mit dem Namen *str* gesucht. Wie bei *Find* wird bei erfolgreicher Suche der Zeiger auf den gefundenen Knoten und sonst *NIL* zurückgeliefert.

Typgebundene Prozeduren

Da *Root* als Erweiterung von *BasicTypes.COLLECTION* (s.u.) definiert ist, werden auch die typgebundenen Prozeduren von *COLLEC-*

TION redefiniert. Diese sind *Add*, *Remove*, *nbElements*, und *Do*. *Add*, *Remove* und *Do* entsprechen den gewöhnlichen Prozeduren *Add*, *Remove* und *DoForward*. *nbElements* liefert die Anzahl der Knoten des Baumes.

Beispielprogramm:

Dieses Programm verwaltet einen einfachen Geburtstagskalender.

```
MODULE Birthday;

IMPORT AVL, io;

TYPE
  Person = POINTER TO PersonDesc;
  PersonDesc = RECORD (AVL.SNode)
    birthday: ARRAY 20 OF CHAR;
  END;

VAR
  root: AVL.SRoot;
  node: AVL.NodePtr;
  new: Person;
  eingabe: ARRAY 20 OF CHAR;
  name: AVL.String;

BEGIN
  AVL.SInit(root);
  REPEAT
    io.WriteLn;
    io.WriteString("  N: Neue Person\n");
    io.WriteString("  G: Geburtstag ausgeben\n");
    io.WriteString("  E: Ende\n");
    io.WriteString("Eingabe: ");
    io.ReadString(eingabe);
    io.WriteLn;
    IF (eingabe="n") OR (eingabe="g") THEN
      io.WriteString("Name der Person: ");
      io.ReadString(name);
      IF eingabe="n" THEN
        NEW(new); new.name := name;
        io.WriteString("Geburtstag: ");
        io.ReadString(new.birthday);
        IF ~ AVL.Add(root,new) THEN
```

```
          io.WriteString("Person existiert bereits!\n");
        END;
      ELSE
        node := AVL.SFind(root,name);
        IF node=NIL THEN
          io.WriteString("Person nicht gefunden!\n");
        ELSE
          io.WriteString("Geburtstag: ");
          io.WriteString(node(Person).birthday);
          io.WriteLn;
        END;
      END;
    END;
  UNTIL eingabe="e";
END Birthday.
```

AVLTrees

```
DEFINITION AVLTrees;

IMPORT BT := BasicTypes, BI := BinaryTrees;

TYPE
  Node = POINTER TO NodeDesc;
  NodeDesc = RECORD (BI.NodeDesc) END;
  Root = POINTER TO RootDesc;
  RootDesc = RECORD (BI.RootDesc)
    PROCEDURE (root:Root) Add(node: BT.ANY);
    PROCEDURE (root:Root) Remove(node: BT.ANY);
  END;

TYPE
  String = ARRAY  OF CHAR;
  SNode = POINTER TO SNodeDesc;
  SNodeDesc = RECORD (NodeDesc)
    name : String;
    PROCEDURE (a:SNode) Compare(
                             b: BI.Node): LONGINT;
    PROCEDURE (a:SNode) Find(
                             root: BI.Root): LONGINT;
```

```
  END;
  SRoot = POINTER TO SRootDesc;
  SRootDesc = RECORD (RootDesc)
    PROCEDURE (root:SRoot) SFind(
                            str: String): SNode;
  END;

PROCEDURE Create(): Root;
PROCEDURE SCreate(): SRoot;

END AVLTrees.
```

Dieses Modul ist dem Modul *AVL* sehr ähnlich. Der Unterschied ist, daß es ein Oberon-2 Modul ist, es enthält also vor allem typgebundene Prozeduren.

Die Prozeduren zum Vergleichen und Suchen von Elementen sind hier typgebunden zum Knotentyp *Node*. Sie werden von *BinaryTrees.Node* geerbt und werden daher in dieser Definitionsdatei nicht aufgelistet. *BinaryTrees* wird weiter unten in diesem Kapitel beschrieben. Module, die *AVLTrees* verwenden, müssen die typgebundenen Prozeduren *Compare* und *Find* redefinieren.

PROCEDURE Create(): Root

Diese Prozedur erzeugt einen neuen, leeren AVL-Baum.

PROCEDURE (root: Root) Add(node: BT.ANY);

root.Add(node) fügt den Knoten *node* (vom Typ *Node*) in den Baum ein. Der Erfolg des Einfügens kann mit *root.addOk* (geerbt von *BinaryTrees.RootDesc*) geprüft werden.

PROCEDURE (root: Root) Remove(node: BT.ANY);

Das Element *node* (vom Typ *Node*) wird aus dem Baum

entfernt. Der Erfolg kann hier mit *root.remOk* (geerbt von *BinaryTrees.RootDesc*) geprüft werden.

Find, nbElements, isEmpty, Do und DoBackward

Diese typgebunden Prozeduren erbt *Root* von *BinaryTrees.Root* ohne sie zu redefinieren. Sie können auch für AVL-Bäume verwendet werden. Innerhalb der bei *Do* bzw. *DoBackward* angegebenen Prozedur dürfen *Add* und *Remove* nicht aufgerufen werden.

Zeichenketten-AVL-Bäume

Wie *AVL* bietet auch *AVLTrees* spezielle Typen und Prozeduren für Bäume, die nach Zeichenketten sortiert sind. Ein solcher Baum hat als Wurzel eine Struktur *SRootDesc* und als Blätter Erweiterungen von *SNodeDesc*. Die speziellen Prozeduren sind:

PROCEDURE SCreate(): SRoot;

Wie bei Create wird hier ein neuer, leerer Zeichenketten-Baum erzeugt.

PROCEDURE (root: SRoot) SFind(str: String): SNode;

Der Knoten mit dem Namen *str* wird gesucht und ein Zeiger darauf oder *NIL* zurückgeliefert.

Alle anderen typgebundenen Prozeduren können unverändert auch für Zeichenketten-Bäume verwendet werden.

BasicTypes

```
DEFINITION BasicTypes;

TYPE

  ANY = POINTER TO ANYDesc;
  ANYDesc = RECORD
  END;

  COLLECTION = POINTER TO COLLECTIONDesc;
  COLLECTIONDesc = RECORD (ANYDesc)
    PROCEDURE (c:COLLECTION) Add(x: ANY);
    PROCEDURE (c:COLLECTION) Remove(x: ANY);
    PROCEDURE (c:COLLECTION) nbElements(): LONGINT;
    PROCEDURE (c:COLLECTION) isEmpty(): BOOLEAN;
    PROCEDURE (c:COLLECTION) Do(p: DoProc;
                                 par: ANY);
  END;

  DoProc = PROCEDURE(x, par: ANY);

  COMPAREABLE = POINTER TO COMPAREABLEDesc;
  COMPAREABLEDesc = RECORD (ANYDesc)
    PROCEDURE (a:COMPAREABLE) Compare(
                           b: COMPAREABLE): LONGINT;
    PROCEDURE (a:COMPAREABLE) Equal(
                           b: COMPAREABLE): BOOLEAN;
    PROCEDURE (a:COMPAREABLE) Less(
                           b: COMPAREABLE): BOOLEAN;
    PROCEDURE (a:COMPAREABLE) LessOrEqual(
                           b: COMPAREABLE): BOOLEAN;
    PROCEDURE (a:COMPAREABLE) Higher(
                           b: COMPAREABLE): BOOLEAN;
    PROCEDURE (a:COMPAREABLE) HigherOrEqual(
                           b: COMPAREABLE): BOOLEAN;
  END;

  GROUP = POINTER TO GROUPDesc;
```

```
  GROUPDesc = RECORD (COMPAREABLEDesc)
    PROCEDURE (m:GROUP) Add(n: GROUP): GROUP;
    PROCEDURE (m:GROUP) Neg(): GROUP;
    PROCEDURE (m:GROUP) Sub(n: GROUP): GROUP;
    PROCEDURE (m:GROUP) Norm(): LONGREAL;
  END;

  RING = POINTER TO RINGDesc;
  RINGDesc = RECORD (GROUPDesc)
    PROCEDURE (m:RING) Mul(n: RING): RING;
    PROCEDURE (m:RING) Sqr(): RING;
  END;

  FIELD = POINTER TO FIELDDesc;
  FIELDDesc = RECORD (RINGDesc)
    PROCEDURE (m:FIELD) Inv(): FIELD;
    PROCEDURE (m:FIELD) InvAllowed(): BOOLEAN;
    PROCEDURE (m:FIELD) Div(n: FIELD): FIELD;
  END;

  DynString = POINTER TO ARRAY OF CHAR;

END BasicTypes.
```

Dieses Modul definiert eine Reihe von grundlegenden Typen. Diese Typen sind nicht für die direkte Anwendung gedacht, sondern dienen dazu, Typen mit ähnlichen Eigenschaften zu gruppieren. Für die Typen werden hier typgebundene Prozeduren definiert, die selbst noch nichts sinnvolles tun, sie führen zu einem Programmabbruch. Erweiterungen der hier definierten Typen können die Prozeduren jedoch redefinieren, so daß sie etwas sinnvolles tun. Alle solchen Erweiterungen sind kompatibel zu den hier definierten Grundtypen. Dadurch können mit den hier definierten Typen abstrakte Prozeduren geschrieben werden, die später mit verschiedenen konkreten Typen arbeiten.

Die Typen haben im Einzelnen die folgenden Funktionen:

ANY

ANY ist der allgemeinste Zeigertyp. Er zeigt auf einen leeren Recordtyp. Alle anderen Recordtypen sollten als Erweiterungen von *ANYDesc* definiert werden. So sind alle Zeiger auf Records zuweisungskompatibel zu Variablen des Typs *ANY.* Dies wird in vielen Modulen ausgenützt, deren Prozeduren vom Benutzer beliebig erweiterte Typen erhalten sollen.

COLLECTION

Dieser Typ ist eine Verallgemeinerung für Datenstrukturen, die Objekte sammeln und zusammen speichern. Dies sind beispielsweise Listen oder Bäume. Eine *COLLECTION* stellt die folgenden typgebundenen Prozeduren zu Verfügung:

PROCEDURE (c: COLLECTION) Add(x: ANY);

Das Objekt *x* wird in die *COLLECTION* aufgenommen.

PROCEDURE (c: COLLECTION) Remove(x: ANY);

Das Objekt *x*, das in der *COLLECTION* enthalten sein muß, wird aus ihr entfernt.

PROCEDURE (c: COLLECTION) nbElements(): LONGINT;
PROCEDURE (c: COLLECTION) isEmpty(): BOOLEAN;

Das *c.nbElements()* liefert die Anzahl an Objekten, die die *COLLECTION c* enthält. *isEmpty()* liefert *TRUE*, wenn *c* keine Objekte enthält. *isEmpty* ist in *BasicTypes* bereits als *c.nbElements=0* implementiert. Es sollte nur dann redefiniert werden, wenn es deutlich effizienter als *nbElements* implementiert werden kann.

PROCEDURE (c: COLLECTION) Do(p: DoProc; par: ANY);
DoProc = PROCEDURE(x,par: ANY);

Die Prozedur *p* wird mit allen Objekten, die in *c* gespeichert sind, als Parameter *x* aufgerufen. Der Parameter *par* wird an *p* weitergegeben. Hier können beliebige Erweiterungen des Typs *ANYDesc* übergeben werden, die von *p* benötigte Informationen enthalten.

COMPAREABLE

Von diesem Typ sollten Objekte sein, die miteinander vergleichbar sind. Zwischen zwei solchen Objekten *a* und *b* muß immer genau eine der drei Beziehungen bestehen: *a* ist kleiner als *b*, *a* ist gleich *b* oder *a* ist größer als *b*.

Die typgebundenen Prozeduren der Erweiterungen des Typs *COMPAREABLE* sind:

PROCEDURE (a: COMPAREABLE) Compare
(b: COMPAREABLE): LONGINT;

Dies ist die einzige Prozedur, die in Erweiterungen von *COMPAREABLE* redefiniert werden muß. Sie vergleicht *a* und *b*. Das Ergebnis darf folgende Werte annehmen:

```
a.Compare(b) < 0, falls a^ < b^
a.Compare(b) = 0, falls a^ = b^
a.Compare(b) > 0, falls a^ > b^
```

Equal, Less, LessOrEqual, Higher, HigherOrEqual

Diese Prozeduren sind in *BasicTypes* schon mit Hilfe von *Compare* implementiert. Sie brauchen also nicht redefiniert zu werden. Können sie jedoch effizienter implementiert werden als *Compare*, so sollten sie dennoch redefiniert werden.

GROUP

Mit diesem Typ werden Elemente einer mathematischen Gruppe definiert (genaueres hierzu ist in fast jedem Buch über Algebra zu finden). Die typgebundenen Prozeduren haben folgende Bedeutung:

PROCEDURE (m: GROUP) Add(n: GROUP): GROUP;

Add ist die in der Gruppe definierte Verknüpfung. Dabei müssen folgende Bedingungen (Gruppenaxiome) erfüllt sein:

1. Add muß assoziativ sein, es muß also *a.Add(b.Add(c))* und *a.Add(b).Add(c)* für alle *a*, *b*, *c* dasselbe ergeben (hier wurde im zweiten Ausdruck die Möglichkeit ausgenutzt, in Amiga Oberon Funktionsaufrufe innerhalb von Designatoren zu benutzen. In gewöhnlichem Oberon-2 muß man sich hier mit einer temporären Variablen helfen).
2. Es muß ein neutrales Element *z* geben, so daß *a.Add(z).Equal(a)* für alle *a* gilt.
3. Zu jedem *a* muß es ein inverses Element *a.Neg()* geben, wobei *a.Add(a.Neg()).Equal(z)* für alle *a* gilt.

PROCEDURE (m: GROUP) Neg(): GROUP;

Das Ergebnis ist das inverse Element zu *m*.

PROCEDURE (m: GROUP) Sub(n: GROUP): GROUP;

m.Sub(n) ist lediglich eine Abkürzung für *m.Add(n.Neg())*. *Sub* ist in *BasicTypes* bereits so implementiert, es braucht nur dann redefiniert zu werden, wenn es dadurch deutlich effizienter wird.

PROCEDURE (m: GROUP) Norm(): LONGREAL;

Bei vielen Gruppen ist es sinnvoll, eine Norm zu definieren. Dies ist z.B. bei den Vektorräumen $\mathbf{R}^n$ der Fall. Die *m.Norm()*

muß genau dann 0 liefern, wenn *m* das neutrale Element *z* der Gruppe ist.

RING

Mit Erweiterungen dieses Typs werden Elemente eines mathematischen Rings beschrieben. Ringe sind spezielle Gruppen mit einer Zusätzlichen Verknüpfung. Für sie müssen also die oben beschriebenen Gruppenaxiome auch gelten. Außer den von *GROUP* geerbten Prozeduren enthält der Typ *RING* die folgenden typgebundenen Prozeduren:

PROCEDURE (m: RING) Mul(n: RING): RING;

Dies ist die neue Verknüpfung in der Gruppe. Für sie und *Add* muß dabei gelten:

1. *Mul* muß assoziativ sein, es muß also *a.Mul(b.Mul(c))* und *a.Mul(b).Mul(c)* für alle *a*, *b*, *c* dasselbe ergeben.
2. Es muß stets *a.Mul(b.Add(c))* dasselbe ergeben wie *a.Mul(b).Add(a.Mul(c))* (erstes Distributivgesetz).
3. Es muß stets *a.Add(b).Mul(c)* dasselbe ergeben wie *a.Mul(c).Add(b.Mul(c))* (zweites Distributivgesetz).

PROCEDURE (m: RING) Sqr(): RING;

Diese Prozedur ist als *m.Mul(m)* bereits implementiert. Sie sollte nur dann redefiniert werden, wenn das Anwenden von *Mul* auf zwei gleiche Objekte effizienter implementiert werden kann.

FIELD

Objekte, die einen mathematischen Körper (engl. Field) bilden, sollten als Erweiterungen von *FIELD* definiert werden. Für dies Objekte sind die beiden Verknüpfungen des Rings, *Add* und *Mul*, definiert. Außer den Ringaxiomen muß hier gelten:

1. *Add* muß kommutativ sein, also *a.Add(b)* muß stets dasselbe ergeben wie *b.Add(a)*.
2. *Mul* muß kommutativ sein, also *a.Mul(b)* muß stets dasselbe ergeben wie *b.Mul(a)*.
3. Es muß ein Element *e* geben, so daß *a.Mul(e).Equal(a)* für alle a gilt.
4. Für alle *x* ungleich *z* (*z* ist das neutrale Element der Gruppe mit *Add)* muß es ein inverses Element *x.Inv()* geben, so daß *x.Mul(x.Inv()).Equal(z)* gilt.

Außer den typgebundenen Prozeduren des Rings kennt ein Körper folgende Prozeduren:

PROCEDURE (m: FIELD) Inv(): FIELD;

Ergebnis ist das inverse Element zu *m*. *m.Inv()* darf nur aufgerufen werden, wenn *m.InvAllowed() TRUE* ergibt.

PROCEDURE (m: FIELD) InvAllowed(): BOOLEAN;

Ergebnis von *m.InvAllowed()* ist *TRUE*, wenn *~m.Equal(z)*.

PROCEDURE (m: FIELD) Div(n: FIELD): FIELD;

Diese Prozedur ist bereits als *m.Mul(n.Inv())* implementiert. Kann dies effizienter berechnet werden, sollte *Div* redefiniert werden.

DynSTRING

Dies ist ein allgemeiner Typ für beliebig lange Zeichenketten.

BigSets

```
DEFINITION BigSets;

TYPE
  BigSet = POINTER TO BigSetDesc;
  BigSetDesc = RECORD (BasicTypes.ANYDesc)
    nbElements : LONGINT;
    PROCEDURE (b:BigSet) Copy(): BigSet;
    PROCEDURE (b:BigSet) Incl(e: LONGINT);
    PROCEDURE (b:BigSet) Excl(e: LONGINT);
    PROCEDURE (b:BigSet) In(e: LONGINT): BOOLEAN;
    PROCEDURE (b:BigSet) Complement(): BigSet;
    PROCEDURE (b:BigSet) Difference(
                              c: BigSet): BigSet;
    PROCEDURE (b:BigSet) Intersection(
                              c: BigSet): BigSet;
    PROCEDURE (b:BigSet) SymmetricDifference(
                              c: BigSet): BigSet;
    PROCEDURE (b:BigSet) Union(
                              c: BigSet): BigSet;
  END;

PROCEDURE Create(nbElements: LONGINT): BigSet;

END BigSets.
```

Mit Hilfe dieses Moduls können Mengen beliebiger Länge bearbeitet werden. Die Standardmengentypen (*SHORTSET*, *SET* und *LONGSET*) sind auf wenige Elemente beschränkt. Dies gilt für *BigSets* nicht.

Auf die Mengen kann mit den folgenden Prozeduren zugegriffen werden:

PROCEDURE Create(nbElements: LONGINT): BigSet;

Es wird ein leeres Mengenobjekt mit Platz für *nbElements* Ele-

menten erzeugt. Dem Recordelement *nbElements* wird die Größe der Menge zugewiesen.

PROCEDURE (b: BigSet) Copy(): BigSet;

Der Aufruf *b.Copy()* erzeugt eine neue Menge mit dem gleichen Inhalt wie *b*.

PROCEDURE (b: BigSet) Incl(e: LONGINT);
PROCEDURE (b: BigSet) Excl(e: LONGINT);

Das Element *e* wird von diesen Prozeduren in die Menge aufgenommen (*Incl*) bzw. aus ihr entfernt (*Excl*). Diese Funktionen entsprechen den Standardprozeduren *INCL* und *EXCL*.

PROCEDURE (b: BigSet) In(e: LONGINT): BOOLEAN;

Es wird geprüft, ob *e* in *b* enthalten ist. Dies entspricht dem Standardoperator *IN*.

PROCEDURE (b: BigSet) Complement(): BigSet;

b.Complement() erzeugt eine neue Menge, wobei für alle Elemente *e* zwischen *0* und *b.nbElements-1* gilt *b.In(e) = ~b.Complement().In(e)*.

PROCEDURE (b: BigSet) Difference(c: BigSet): BigSet
PROCEDURE (b: BigSet) Intersection(c: BigSet): BigSet
PROCEDURE (b: BigSet) SymmetricDifference(
c: BigSet): BigSet
PROCEDURE (b: BigSet) Union(c: BigSet): BigSet

Diese Prozeduren liefern jeweils eine neue Menge. Sie enthält die Differenz, die Schnittmenge, die symmetrische Differenz bzw. die Vereinigung von *b* und *c*. Dies entspricht den Operatoren "-", "*", "/" und "+" bei gewöhnlichen Mengentypen.

BinaryTrees

```
DEFINITION BinaryTrees;

IMPORT BT := BasicTypes;

TYPE
  Node = POINTER TO NodeDesc;
  NodeDesc = RECORD (BT.COMPAREABLEDesc)
    l : Node;
    r : Node;
    PROCEDURE (a:Node) Find(root: Root): LONGINT;
  END;
  Root = POINTER TO RootDesc;
  RootDesc = RECORD (BT.COLLECTIONDesc)
    root : Node;
    addOk : BOOLEAN;
    remOk : BOOLEAN;
    PROCEDURE (root:Root) Add(node: BT.ANY);
    PROCEDURE (root:Root) Find(): Node;
    PROCEDURE (root:Root) Remove(node: BT.ANY);
    PROCEDURE (tree:Root) nbElements(): LONGINT;
    PROCEDURE (tree:Root) isEmpty(): BOOLEAN;
    PROCEDURE (tree:Root) Do(p: BT.DoProc;
                         par: BT.ANY);
    PROCEDURE (root:Root) DoBackward(
                         proc: BT.DoProc; par: BT.ANY);
    PROCEDURE (root:Root) Dispose;
  END;

PROCEDURE Create(): Root;

END BinaryTrees.
```

Dieses Modul stellt Typen für sortierte Binärbäume zur Verfügung. Binärbäume ermöglichen einen schnellen Zugriff und ein schnelles Einfügen von Elementen. Nur beim sortierten Einfügen von Elementen kann die Effizienz dieser Bäume sehr schlecht werden. Dann sollten stattdessen AVL-Bäume verwendet werden. Das Modul *AVLTrees*

erweitert die Typen von *BinaryTrees* so, daß damit AVL-Bäume erzeugt werden (s.o.).

Um dieses Modul verwenden zu können, muß eine Erweiterung des Typs *NodeDesc* definiert werden. Diese Erweiterung muß die Daten, die in dem Baum gespeichert werdenen sollen, enthalten. Zudem müssen die typgebundenen Prozeduren *Compare* und *Find* redefiniert werden. *Compare* wird von *BasicTypes.COMPAREABLE* geerbt (s.o.) und muß so implementiert werden wie von *BasicTypes* vorgeschrieben. *Find* wird für das Suchen nach Knoten in dem Baum benötigt. Es muß dabei gelten:

```
a.Find(root) < 0, falls a^ < gesuchtes Element
a.Find(root) = 0, falls a^ = gesuchtes Element
a.Find(root) > 0, falls a^ > gesuchtes Element
```

Der Parameter *root* kann eine Erweiterung von *Root* sein, die Informationen über das gesuchte Element enthält, die zu diesem Vergleich nötig sind.

Die Prozeduren dieses Moduls im Einzelnen:

PROCEDURE Create(): Root;

Create() erzeugt einem neuen, leeren Binärbaum.

PROCEDURE (root: Root) Add(node: BT.ANY);

Der Knoten *node* (vom Typ *Node*) wird in dem Baum eingefügt. Konnte der Knoten nicht eingefügt werden, weil schon ein gleicher Knoten im Baum enthalten ist, wird *root.addOk* auf *FALSE* gesetzt, ansonsten auf *TRUE*.

PROCEDURE (root: Root) Find(): Node;

Es wird mit Hilfe der redefinierten Prozedur *Node.Find* nach ei-

nem Knoten gesucht. Wurde der Knoten gefunden, so liefert *Find* einen Zeiger auf diesen Knoten, sonst ist das Ergebnis *NIL*.

PROCEDURE (root: Root) Remove(node: BT.ANY);

Der Knoten *node* (vom Typ *Node*) wird aus dem Baum entfernt. Der Erfolg dieser Operation wird in *root.remOk* angezeigt. Es ist *FALSE*, wenn der Knoten nicht in dem Baum enthalten war, sonst *TRUE*.

PROCEDURE (root: Root) nbElements(): LONGINT;

Ergebnis ist die Anzahl der Knoten des Baumes.

PROCEDURE (root: Root) isEmpty(): BOOLEAN;

Ergebnis ist *TRUE* wenn der Baum keinen Knoten enthält.

PROCEDURE (root: Root) Do(p: BT.DoProc; par: BT.ANY);

p wird mit jedem Knoten des Baumes in Infix-Reihenfolge (von links nach rechts) als Parameter aufgerufen. *par* wird dabei, wie in *BasicTypes* vorgeschrieben, als zweiter Parameter übergeben. Innerhalb der Prozedur *p* darf *Add* und *Remove* nicht aufgerufen werden.

PROCEDURE (root: Root) DoBackward(p: BT.DoProc;
par: BT.ANY);

p wird mit wie bei *Do* mit jedem Knoten des Baumes als Parameter aufgerufen. Dabei wird der Baum jedoch rückwärts durchlaufen. Auch hier darf innerhalb der Prozedur *p Add* und *Remove* nicht aufgerufen werden.

PROCEDURE (root: Root) Dispose;

Diese Prozedur löscht den gesamten Baum. Wird mit Garbage-Collector gearbeitet, so wird einfach die Wurzel auf *NIL* gesetzt. Ohne Garbage-Collector wird dagegen der Speicher jedes Elementes mit *DISPOSE* freigegeben, auf die Elemente darf dann also auch über andere Referenzen nicht mehr zugegriffen werden.

FArrays

```
DEFINITION FArrays;

IMPORT BT := BasicTypes;

TYPE
  FArray = POINTER TO FArrayDesc;
  FArrayDesc = RECORD (BT.COLLECTIONDesc)
    lower,upper : LONGINT;
    PROCEDURE (a:FArray) Put(v: BT.ANY;
                              at: LONGINT);
    PROCEDURE (a:FArray) Get(at: LONGINT): BT.ANY;
    PROCEDURE (a:FArray) Resize(
                   minindex, maxindex: LONGINT);
    PROCEDURE (a:FArray) Add(x: BT.ANY);
    PROCEDURE (a:FArray) Remove(x: BT.ANY);
    PROCEDURE (a:FArray) nbElements(): LONGINT;
    PROCEDURE (a:FArray) Do(p: BT.DoProc;
                             par: BT.ANY);
  END;

PROCEDURE Create(minindex,
                 maxindex: LONGINT): FArray;

END FArrays.
```

Dieses Modul bietet einen flexiblen Feldtyp an. Das Feld kann einen beliebigen Indexbereich besitzen. Zudem kann dieser Bereich nachträglich verändert werden. In dem Feld können beliebige Zeiger auf

Records, genauer auf Erweiterungen von *BasicTypes.ANYDesc*, gespeichert werden.

Der Typ *FArrayDesc* ist als Erweiterung von *BasicTypes.COLLECTIONDesc* definiert, obwohl ein *FArray* keine wirkliche *COLLECTION* ist. Die typgebundene Prozedur *Add* darf hier nicht aufgerufen werden, da sie nichts sinnvolles tun kann. Sie führt zu einem Abbruch. Die Prozeduren von FArrays:

PROCEDURE Create(minindex, maxindex: LONGINT): FArray;

Es wird ein neues Feld mit dem Indexbereich *minindex..maxindex* erzeugt. Alle Elemente des Feldes werden auf *NIL* gesetzt. Die Elemente *lower* und *upper* der erzeugten *FArray*-Variablen werden auf *minindex* bzw. *maxindex* gesetzt.

PROCEDURE (a: FArray) Put(v: BT.ANY; at: LONGINT);

Das Feldelement an Position *at* wird auf *v* gesetzt. *at* muß zwischen *a.lower* und *a.upper* (jeweils einschließlich) liegen.

PROCEDURE (a: FArray) Get(at: LONGINT): BT.ANY;

Der Inhalt des Feldelementes an Position *at* wird zurückgegeben. *at* muß zwischen *a.lower* und *a.upper* (jeweils einschließlich) liegen.

PROCEDURE (a: FArray) Resize(
minindex, maxindex: LONGINT): FArray;

Das Feld wird noch oben oder nach unten vergrößert, wenn *minindex<a.lower* bzw. *maxindex>a.upper* ist. Die bisherigen Elemente des Feldes bleiben dabei unberührt. Die eventuell neu hinzugekommenen Elemente werden auf *NIL* gesetzt.

PROCEDURE (a: FArray) Add(x: BT.ANY);

Diese von *BasicTypes.COLLECTION* geerbte Prozedur darf hier nicht verwendet werden.

PROCEDURE (a: FArray) Remove(x: BT.ANY);

Alle Vorkommen von *x* in *a* werden auf *NIL* gesetzt.

PROCEDURE (a: FArray) nbElements(): LONGINT;

Das Ergebnis ist die Anzahl der Elemente des Feldes.

PROCEDURE (a: FArray) Do(p: BT.DoProc; par: BT.ANY);

p wird mit jedem Element des Feldes als Parameter (auch mit den Elementen, die *NIL* sind) aufgerufen. *par* wird dabei, wie in *BasicTypes* vorgeschrieben, als zweiter Parameter übergeben.

LinkedLists

```
DEFINITION LinkedLists;

IMPORT BT := BasicTypes;

TYPE
  Node = POINTER TO NodeDesc;
  List = POINTER TO ListDesc;
  NodeDesc = RECORD (BT.ANYDesc)
    next,prev : Node;
    PROCEDURE (n:Node) Remove;
    PROCEDURE (x:Node) AddBefore(n: Node);
    PROCEDURE (x:Node) AddBehind(n: Node);
  END;
  ListDesc = RECORD (BT.COLLECTIONDesc)
   head : Node;
   tail : Node;
```

```
    PROCEDURE (list:List) Init;
    PROCEDURE (list:List) AddHead(n: Node);
    PROCEDURE (list:List) AddTail(n: Node);
    PROCEDURE (list:List) RemHead(): Node;
    PROCEDURE (list:List) RemTail(): Node;
    PROCEDURE (list:List) Add(x: BT.ANY);
    PROCEDURE (list:List) Remove(x: BT.ANY);
    PROCEDURE (list:List) nbElements(): LONGINT;
    PROCEDURE (list:List) Do(p: BT.DoProc;
                               par: BT.ANY);
    PROCEDURE (list:List) DoBackward(p: BT.DoProc;
                                       par: BT.ANY);
  END;

PROCEDURE Create(): List;

END LinkedLists.
```

Dieses Modul definiert Typen und die dazugehörenden typgebundenen Prozeduren für doppelt verkettete Listen. Als Listenelemente müssen Erweiterungen von *NodeDesc* definiert werden. Auf das erste bzw. letzte Element einer Liste kann über *Lists.head* bzw. *Lists.tail* zugegriffen werden. Mit *Node.next* und *Node.prev* kann der Nachfolger bzw. Vorgänger eines Knotens gefunden werden.

Zum Arbeiten mit diesen Listen bietet *LinkedLists* folgende Prozeduren:

PROCEDURE Create(): List;

Es wird eine neue leere Liste erzeugt.

PROCEDURE (list: List) Init;

Die Liste *list* wird gelöscht. Sie ist danach leer.

PROCEDURE (list: List) AddHead(n: Node);
PROCEDURE (list: List) AddTail(n: Node);

Das Element *n* wird am Anfang (Head) bzw. Ende (Tail) von *list* angefügt.

PROCEDURE (list: List) RemHead(): Node;
PROCEDURE (list: List) RemTail(): Node;

Das Element am Anfang (RemHead) bzw. am Ende (RemTail) von *list* wird aus der Liste entfernt und als Ergebnis zurückgegeben. Ist *list* leer, so liefern diese Prozeduren als Ergebnis *NIL*.

PROCEDURE (list: List) Add(x: BT.ANY);

Das Element *x* wird in die Liste eingefügt. Dabei ist nicht festgelegt, wo es eingefügt wird.

PROCEDURE (list: List) Remove(x: BT.ANY);

Das Element *x*, das sich in der Liste befinden muß, wird aus ihr entfernt.

PROCEDURE (list: List) nbElements(): LONGINT;
PROCEDURE (list: List) isEmpty(): BOOLEAN;

list.nbElements() liefert die Anzahl der Elemente einer Liste. *list.isEmpty()* ist gleichbedeutend mit *list.nbElements()=0.*

PROCEDURE (list: List) Do(p: BT.DoProc; par: BT.ANY);

Die Liste wird vorwärts durchlaufen wobei *p* mit jedem Element der Liste als ersten Parameter aufgerufen wird. *par* wird, wie in *BasicTypes* vorgeschrieben, als zweiter Parameter übergeben.

PROCEDURE (list: List) DoBackward(p: BT.DoProc;
par: BT.ANY);

Wie bei *Do* wird *p* mit jedem Element der Liste aufgerufen, die Liste wird hier jedoch rückwärts, als von *list.tail* bis *list.head* durchlaufen.

PROCEDURE (n: Node) Remove;

Der Knoten *n* wird aus der Liste, in die er zuvor eingefügt wurde, entfernt.

PROCEDURE (x: Node) AddBefore(n: Node);
PROCEDURE (x: Node) AddBehind(n: Node);

Der Knoten *n* wird vor (AddBefore) bzw. hinter (AddBehind) dem Knoten *x* in die Liste eingefügt, in der sich *x* befindet.

Lists

```
DEFINITION Lists;

IMPORT BT := BasicTypes;

TYPE
  NodePtr = POINTER TO Node;
  Node = RECORD (BT.ANYDesc)
    next : NodePtr;
    prev : NodePtr;
  END;
  ListPtr = POINTER TO List;
  List = RECORD (BT.COLLECTIONDesc)
    head : NodePtr;
    tail : NodePtr;
    PROCEDURE (list:ListPtr) Add(x: BT.ANY);
    PROCEDURE (list:ListPtr) Remove(x: BT.ANY);
    PROCEDURE (list:ListPtr) nbElements(): LONGINT;
```

```
    PROCEDURE (list:ListPtr) isEmpty(): BOOLEAN;
    PROCEDURE (list:ListPtr) Do(p: BT.DoProc;
                                par: BT.ANY);
  END;
  DoProc = PROCEDURE(n: NodePtr);

PROCEDURE Init(VAR list: List);
PROCEDURE AddHead(VAR list: List; n: NodePtr);
PROCEDURE AddTail(VAR list: List; n: NodePtr);
PROCEDURE Remove(VAR list: List; n: NodePtr);
PROCEDURE RemHead(VAR list: List): NodePtr;
PROCEDURE RemTail(VAR list: List): NodePtr;
PROCEDURE AddBefore(VAR list: List;
                    n, x: NodePtr);
PROCEDURE AddBehind(VAR list: List;
                    n, x: NodePtr);
PROCEDURE Empty(VAR list: List): BOOLEAN;
PROCEDURE CountElements(VAR list: List): LONGINT;
PROCEDURE DoForward(VAR list: List; proc: DoProc);
PROCEDURE DoBackward(VAR list: List;
                     proc: DoProc);
PROCEDURE Next(VAR n: NodePtr): BOOLEAN;
PROCEDURE Previous(VAR n: NodePtr): BOOLEAN;
PROCEDURE Succ(VAR n: NodePtr);
PROCEDURE Pred(VAR n: NodePtr);
PROCEDURE Head(VAR list: List): NodePtr;
PROCEDURE Tail(VAR list: List): NodePtr;

END Lists.
```

Wie *LinkedLists* bietet dieses Modul Typen und Prozeduren zum Arbeiten mit doppelt verketteten Listen. Anders als *LinkedLists* stammt *Lists* noch aus der Zeit, als Amiga Oberon die Erweiterungen von Oberon-2 noch nicht unterstützte. Daher sind in *Lists* bis auf die von *BasicTypes.COLLECTIONDesc* geerbten Prozeduren alle Prozeduren nicht typgebunden.

Die Elemente *head* und *tail* von *List* Zeigen auf das erste bzw. letzte

Listenelement. *next* und *prev* von *Node* sind der Nachfolger bzw. der Vorgänger des Knotens. Die Prozeduren sind:

PROCEDURE Init(VAR list: List);

Die Liste wird gelöscht.

PROCEDURE AddHead(VAR list: List; n: NodePtr;
PROCEDURE AddTail(VAR list: List; n: NodePtr;
PROCEDURE AddBefore(VAR list: List; n,x: NodePtr);
PROCEDURE AddBehind(VAR list: List; n,x: NodePtr);

Das Element n wird vorne (*AddHead*), hinten (*AddTail*), vor dem Element *x* (*AddBefore*) bzw. hinter dem Element *x* (*AddBehind*) in die Liste eingefügt.

PROCEDURE Remove (VAR list: List; n: NodePtr;
PROCEDURE RemHead(VAR list: List): NodePtr;
PROCEDURE RemTail(VAR list: List): NodePtr;

Das Element *n* (*Remove*) oder das erste (*RemHead*) bzw. letzte (*RemTail*) Element der Liste wird entfernt. Das Ergebnis von *RemHead* und *RemTail* ist ein Zeiger auf das entfernte Element oder *NIL* bei einer leeren Liste.

PROCEDURE Empty(VAR list: List): BOOLEAN;
PROCEDURE CountElements(VAR list: List): LONGINT;

Empty ist *TRUE* für eine leere Liste. *CountElements* ergibt die Anzahl der Elemente der Liste.

PROCEDURE DoForward(VAR list: List; proc: DoProc);
PROCEDURE DoBackward(VAR list: List; proc: DoProc);

Die Liste wird vorwärts bzw. rückwärts durchwandert wobei *proc* mit jedem Element als Parameter aufgerufen wird.

PROCEDURE Next(VAR n: NodePtr): BOOLEAN;
PROCEDURE Prev(VAR n: NodePtr): BOOLEAN;
PROCEDURE Succ(VAR n: NodePtr);
PROCEDURE Pred(VAR n: NodePtr);

n wird auf seinen Nachfolger (*Next*, *Succ*) bzw. Vorgänger (*Prev*, *Pred*) gesetzt. Existiert dieser nicht, wird *n* auf *NIL* gesetzt. *Next* und *Prev* geben dann *FALSE* zurück.

PROCEDURE Head(VAR list: List): NodePtr;
PROCEDURE Tail(VAR list: List): NodePtr;

Ergebnis ist das erste (*Head*) bzw. das letzte (*Tail*) Element der Liste oder *NIL* bei einer leeren Liste.

Add, Remove, nbElements, isEmpty, Do

Diese typgebundenen Prozeduren tun exakt das, was sie nach der Definition in *BasicTypes* tun sollen und was ihre Namensvettern aus *LinkedLists* tun.

Queues

```
DEFINITION Queues;

IMPORT LL := LinkedLists;

TYPE
 Queue = POINTER TO QueueDesc;
 QueueDesc = RECORD (LL.ListDesc)
   PROCEDURE (q:Queue) Put(x: Node);
   PROCEDURE (q:Queue) Get(): Node;
 END;
 Node = POINTER TO NodeDesc;
 NodeDesc = RECORD (LL.NodeDesc)
 END;
```

```
PROCEDURE Create(): Queue;

END Queues.
```

Dieses Modul stellt den generischen Daten Schlange (Queue, FIFO-Liste) zur Verfügung. Die Elemente, die in dieser Schlange gespeichert werden sollen, müssen Erweiterungen von *Queues.Node* sein.
Die Prozeduren aus Queues:

PROCEDURE Create(): Queue;

Es wird eine neue, leere Schlange erzeugt.

PROCEDURE (q: Queue) Put(x: Node);

Das Element *x* wird hinten an die Schlange angefügt.

PROCEDURE (q: Queue) Get(): Node;

Das erste Element aus der Schlange wird entfernt und zurückgegeben. War die Schlange leer, so ist das Ergebnis *NIL*.

Stacks

```
DEFINITION Stacks;

IMPORT LL := LinkedLists;

TYPE
  Stack = POINTER TO StackDesc;
  StackDesc = RECORD (LL.ListDesc)
    PROCEDURE (s:Stack) Push(x: Node);
    PROCEDURE (s:Stack) Pop(): Node;
    PROCEDURE (s:Stack) Top(): Node;
  END;
  Node = POINTER TO NodeDesc;
```

```
  NodeDesc = RECORD (LL.NodeDesc)
  END;

PROCEDURE Create(): Stack;

END Stacks.
```

Dieses Modul stellt den generischen Datentyp Stapel (Stack, LIFO-Liste) zur Verfügung. Die Elemente, die auf dem Stapel gespeichert werden sollen, müssen als Erweiterungen von *Stacks.Node* definiert werden. Die Prozeduren von *Stacks*:

PROCEDURE Create(): Stack;

Es wird ein neuer, leerer Stapel erzeugt.

PROCEDURE (s: Stack) Push(x: Node);

Das Element *x* wird oben auf den Stapel gelegt.

PROCEDURE (s: Stack) Pop(): Node;

Das oberste Element des Stapels wird entfernt und zurückgegeben. Ist der Stapel leer, so ist das Ergebnis *NIL.*

PROCEDURE (s: Stack) Top(): Node;

Das oberste Element des Stapels wird zurückgegeben, aber nicht vom Stapel genommen. Bei einem leeren Stapel ist das Ergebnis *NIL*.

UntracedAVL

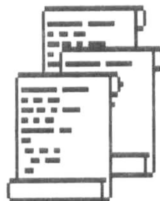
Dieses Modul entspricht dem Modul *AVL* bis auf zwei Ausnahmen: Alle Zeiger sind als *UNTRACED* definiert und *Root* ist keine Erweite-

rung von *COLLECTION*. *UntracedAVL* kann verwendet werden, wenn man den Garbage-Collector nicht benutzen kann oder möchte.

UntracedLists

Dieses Modul entspricht dem Modul *Lists*. Es sind jedoch alle Zeiger als *UNTRACED* definiert und *List* ist keine Erweiterung von *COLLECTION*. *UntracedLists* kann verwendet werden, wenn man den Garbage-Collector nicht benutzen kann oder möchte.

20. Modulbibliothek: Zeichenkettenbearbeitung



Eine der wichtigsten Datenstrukturen ist eine Zeichenkette (String), da Zeichenketten sehr häufig vorkommen. Amiga Oberon bietet mehrere Module, die den Umgang mit Zeichenketten erleichtern. Sie werden in diesem Kapitel beschrieben.

ASCII

```
DEFINITION ASCII;

CONST
  nul = 00X; soh = 01X; stx = 02X; etx = 03X;
  eot = 04X; enq = 05X; ack = 06X; bel = 07X;
  bs  = 08X; ht  = 09X; lf  = 0AX; vt  = 0BX;
  ff  = 0CX; cr  = 0DX; so  = 0EX; si  = 0FX;
  dle = 10X; dc1 = 11X; dc2 = 12X; dc3 = 13X;
  dc4 = 14X; nak = 15X; syn = 16X; etb = 17X;
  can = 18X; em  = 19X; sub = 1AX; esc = 1BX;
  fs  = 1CX; gs  = 1DX; rs  = 1EX; us  = 1FX;
  sp  = 20X;

  del = 7FX;
  eol = lf;
  eof = fs;
  csi = 9BX;

END ASCII.
```

In *ASCII* werden lediglich eine Reihe an Konstanten definiert, die für die verschiedenen Steuerzeichen des ASCII-Codes stehen. Lediglich *csi* ist ein spezielles Zeichen des Amiga, es wird in [RKM: Devices 91] beschrieben.

Conversions

```
DEFINITION Conversions;

PROCEDURE StringToInt(VAR str: ARRAY OF CHAR;
                      VAR int: LONGINT): BOOLEAN;
PROCEDURE IntToString(int: LONGINT;
                      VAR str: ARRAY OF CHAR;
                      n: INTEGER): BOOLEAN;
PROCEDURE IntToHex(int: LONGINT;
                   VAR str: ARRAY OF CHAR;
                   n: INTEGER): BOOLEAN;
PROCEDURE StrToInt(VAR str: ARRAY OF CHAR;
                   VAR int: LONGINT;
                   base: INTEGER): BOOLEAN;
PROCEDURE IntToStr(int: LONGINT;
                   VAR str: ARRAY OF CHAR;
                   base, width: SHORTINT;
                   fillChar: CHAR): BOOLEAN;
PROCEDURE IntToStringLeft(int: LONGINT;
                          VAR str: ARRAY OF CHAR);

END Conversions.
```

Sehr oft ist es nötig, Integerzahlen in Zeichenketten umzuwandeln. Die entsprechenden Prozeduren bietet dieses Module:

**PROCEDURE StringToInt(VAR str: ARRAY OF CHAR;
VAR int: LONGINT): BOOLEAN;**

Die Zeichenkette *str* wird in eine *LONGINT*-Zahl umgewandelt und in *int* zurückgegeben. *str* darf dabei Dezimalzahlen mit Vorzeichen wie *"-23"* oder Hexadezimalzahlen wie *"4E75H"*. Das Ergebnis ist *FALSE* wenn *str* keine gültige Zahl enthält.

PROCEDURE IntToString(int: LONGINT;
VAR str: ARRAY OF CHAR;
n: INTEGER): BOOLEAN;

Der Integerwert *int* wird in eine *n*-stellige rechtsbündige Dezimalzahl umgewandelt. *str* muß dabei Platz für mindestens *n+2* Zeichen haben. Das Ergebnis ist *FALSE* wenn *int* in so groß ist, daß es nicht mit *n* Dezimalstellen dargestellt werden kann. *IntToString(-12,str,3)* ergibt *"- 12"*.

PROCEDURE IntToHex(int: LONGINT;
VAR str: ARRAY OF CHAR;
n: INTEGER): BOOLEAN;

Der Integerwert *int* wird in eine *n*-stellige Hexadezimalzahl umgewandelt. *str* muß dabei Platz für *n+1* Stellen bieten. Ergebnis ist *FALSE* wenn *int* nicht mit *n* Stellen dargestellt werden kann.

PROCEDURE StrToInt(VAR str: ARRAY OF CHAR;
VAR int: LONGINT;
base: INTEGER): BOOLEAN;

Diese Prozedur wandelt wie *StringToInt* die Zahl in *str* in einen Integerwert um. *base* gibt dabei die Basis des Zahlensystems an. Möchte man etwa eine Oktalzahl in einen Integerwert umwandeln, so muß als *base* der Wert *8* übergeben werden. Ergebnis ist auch hier *FALSE* wenn *str* keine Zahl in dem angegebenen Zahlensystem enthält.

PROCEDURE IntToStr(int: LONGINT;
VAR str: ARRAY OF CHAR;
base, width: SHORTINT;
fillChar: CHAR): BOOLEAN;

Wie *IntToString* wandelt auch *IntToStr* eine Integerzahl in eine

Zeichenkette um. Dabei wird das Zahlensystem mit der Basis *base* verwendet und eine rechtsbündige Zahl mit *width* Stellen erzeugt. Werden nicht alle *width* Stellen benötigt, so werden die vordersten Stellen mit *fillChar* gefüllt. Möchte man also führende Nullen, so sollte als *fillChar "0"* angegeben werden. *str* muß Platz für *width+1* Stellen enthalten. Das Ergebnis ist *FALSE* wenn *int* im angegebenen Zahlensystem nicht mit *n* Stellen dargestellt werden kann.

**PROCEDURE IntToStringLeft(int: LONGINT;
VAR str: ARRAY OF CHAR);**

Die Integerzahl *int* wird in eine linksbündige Dezimalzahl umgewandelt. *str* muß ausreichend lang sein, um die Zahl und evtl. ihr Vorzeichen aufnehmen zu können.

LongRealConversions

```
DEFINITION LongRealConversions;

PROCEDURE StringToReal(VAR str: ARRAY OF CHAR;
                       VAR r: LONGREAL): BOOLEAN;
PROCEDURE RealToString(r: LONGREAL;
                       VAR str: ARRAY OF CHAR;
                       v, n: INTEGER;
                       exp: BOOLEAN): BOOLEAN;

END LongRealConversions.
```

Die Prozeduren dieses Moduls wandeln *LONGREAL*-Zahlen in Zeichenketten um und umgekehrt.

**PROCEDURE StringToReal(VAR str: ARRAY OF CHAR;
VAR r: LONGREAL): BOOLEAN;**

Die Zeichenkette *str* wird in die entsprechende *LONGREAL*-Zahl umgewandelt. Dabei kann *str* aus einem Vorzeichen, den

Vorkommastellen, einem Punkt gefolgt von den Nachkommastellen und einem Exponenten bestehen. Der Exponent besteht aus einem *"E"* gefolgt von einem optionalen Vorzeichen und dem Wert des Exponenten selbst. Das Ergebnis ist *TRUE* wenn *str* eine gültige Zahl enthielt. Beispiele:

str	r	Ergebnis
"+123"	123	TRUE
"12.345E2"	1234.5	TRUE
""	0	TRUE
"-.3e-3"	-0.0003	TRUE
"12-3"	---	FALSE
"23E+4.3"	---	FALSE

PROCEDURE RealToString(r: LONGREAL;
VAR str: ARRAY OF CHAR;
v, n: INTEGER;
exp: BOOLEAN): BOOLEAN;

Die *LONGREAL*-Zahl *r* wird in eine Zeichenkette umgewandelt. Dabei werden *v* Vorkomma- und *n* Nachkommastellen angezeigt. Ist *exp TRUE* so wird auch ein Exponent angezeigt. *str* muß ohne Exponent Platz für mindestens *n+v+3* Zeichen, mit Exponent für *n+v+8* Zeichen bieten. Das Ergebnis ist *TRUE* wenn *r* in dem gewünschten Format dargestellt werden kann.

RealConversions

```
DEFINITION RealConversions;

PROCEDURE StringToReal(VAR str: ARRAY  OF CHAR;
                       VAR r: REAL): BOOLEAN;
PROCEDURE RealToString(r: REAL;
                       VAR str: ARRAY  OF CHAR;
                       v, n: INTEGER;
                       exp: BOOLEAN): BOOLEAN;

END RealConversions.
```

Die Prozeduren aus *RealConversions* entsprechen exakt denen aus *LongRealConversions* (siehe oben). Sie werden daher hier nicht erneut beschrieben.

Strings

```
DEFINITION Strings;

PROCEDURE Length(str: ARRAY OF CHAR): LONGINT;
PROCEDURE Append(VAR s1: ARRAY OF CHAR;
                 s2: ARRAY OF CHAR);
PROCEDURE Occurs(VAR s: ARRAY OF CHAR;
                 search: ARRAY OF CHAR): LONGINT;
PROCEDURE OccursPos(VAR s: ARRAY OF CHAR;
                    search: ARRAY OF CHAR;
                    start: LONGINT): LONGINT;
PROCEDURE Cut(VAR s: ARRAY OF CHAR;
              from, cnt: LONGINT;
              VAR to: ARRAY OF CHAR);
PROCEDURE CapIntl(c: CHAR): CHAR;
PROCEDURE Upper(VAR s: ARRAY OF CHAR);
PROCEDURE UpperIntl(VAR s: ARRAY OF CHAR);
PROCEDURE OverWrite(VAR string: ARRAY OF CHAR;
                    overlay: ARRAY OF CHAR;
                    pos: LONGINT);
PROCEDURE Insert(VAR s: ARRAY OF CHAR;
                 at: LONGINT;
                 str: ARRAY OF CHAR);
PROCEDURE Delete(VAR s: ARRAY OF CHAR;
                 at, cnt: LONGINT);
PROCEDURE AppendChar(VAR s: ARRAY OF CHAR;
                     c: CHAR);
PROCEDURE InsertChar(VAR s: ARRAY OF CHAR;
                     at: LONGINT;
                     c: CHAR);

END Strings.
```

Mit den Prozeduren dieses Modules können Zeichenketten vom Typ *ARRAY OF CHAR* bearbeitet werden.

PROCEDURE Length(str: ARRAY OF CHAR): LONGINT;

Das Ergebnis ist die Anzahl der Zeichen von *str*, also die Anzahl derjenigen Zeichen, die *str* vor dem ersten Vorkommen des String-Ende-Zeichens *0X* enthält oder *LEN(str)*.

**PROCEDURE Append(VAR s1: ARRAY OF CHAR;
s2: ARRAY OF CHAR);**

s2 wird an *s1* hinten angehängt. *s1* muß dabei genügend Platz für die Zeichen von *s1* und *s2* enthalten. Beispiel:

```
s := "Oberon"; Strings.Append(s,"-2");
```

Nach diesen Anweisungen enthält *s* den Text *"Oberon-2"*.

**PROCEDURE Occurs(VAR s: ARRAY OF CHAR;
search: ARRAY OF CHAR): LONGINT;**

s wird nach dem ersten Vorkommen von *search* untersucht. Es wird dabei zwischen Groß- und Kleinschreibung unterschieden. Das Ergebnis ist die Position, ab der *search* in *s* enthalten ist, oder *-1* wenn dies nicht der Fall ist.

**PROCEDURE OccursPos(VAR s: ARRAY OF CHAR;
search: ARRAY OF CHAR;
start: LONGINT): LONGINT;**

s wird nach dem ersten Vorkommen von *search* ab der Position *start* untersucht. Es wird dabei zwischen Groß- und Kleinschreibung unterschieden. Das Ergebnis ist die gefundene Position oder *-1* falls *search* nicht mehr gefunden wurde.

Um alle Vorkommen einer Zeichenkette *search* in einer Zeichenkette *s* zu finden, kann folgendes Programmstück verwendet werden:

```
VAR
  p: LONGINT;
  s,search: ARRAY n OF  CHAR;
BEGIN
  ...
  p := Strings.Occurs(s,search);
  WHILE p >= 0 DO
    io.WriteString("Vorkommen an Position:");
    io.WriteInt(p,4); io.WriteLn;
    p := Strings.OccursPos(s,search,p+1);
  END;
END;
```

PROCEDURE Cut(VAR s: ARRAY OF CHAR;
from, cnt: LONGINT;
VAR to: ARRAY OF CHAR);

Vom Index *from* ausgehend werden *cnt* Zeichen aus *s* nach *to* kopiert. *to* muß dabei ausreichend Platz für *cnt* Zeichen bieten und *s* muß mindestens *from+cnt* Zeichen enthalten. Beispiel: Nach der Anweisung

```
s := "Only Amiga makes it possible!";
Strings.Cut(s,5,5,to);
```

enthält to die Zeichenkette *s* den Text *"Amiga"*.

PROCEDURE CapIntl(c: CHAR): CHAR;

Diese Funktion entspricht der Standardfunktion *CAP*, hier werden jedoch auch internationale Zeichen ("ä","ç",etc.) in Großbuchstaben umgewandelt.

PROCEDURE Upper(VAR s: ARRAY OF CHAR);
PROCEDURE UpperIntl(VAR s: ARRAY OF CHAR);

Alle Zeichen aus *s* werden mit der Standardprozedur *CAP* (bei

Upper) bzw mit *CapIntl* (*UpperIntl*) in Großbuchstaben umgewandelt.

**PROCEDURE Insert(VAR s: ARRAY OF CHAR;
at: LONGINT; str: ARRAY OF CHAR);**

An Position *at* wird *str* in *s* eingefügt. *s* muß dazu noch genügend Platz für die Zeichen von *str* enthalten. So enthält *s* nach der Anweisung

```
s := "Only Amiga makes it possible!";
Strings.Insert(s,11,"Oberon ");
```

die Zeichenkette *"Only Amiga Oberon makes it possible!"*.

**PROCEDURE Delete(VAR s: ARRAY OF CHAR;
at, cnt: LONGINT);**

Von Position *at* ausgehend werden *cnt* Zeichen aus *s* entfernt. So enthält *s* nach der Anweisung

```
s := "Only Amiga Oberon makes it possible!";
Strings.Delete(s,5,6);
```

die Zeichenkette *"Only Oberon makes it possible!"*.

**PROCEDURE AppendChar(VAR s: ARRAY OF CHAR;
c: CHAR);**

Das Zeichen *c* wird hinten an *s* angehängt. *s* muß noch genügend Platz für ein weiteres Zeichen enthalten.

**PROCEDURE InsertChar(VAR s: ARRAY OF CHAR;
at: LONGINT; c: CHAR);**

An Position *at* wird das Zeichen *c* in *s* eingefügt. *s* muß dazu noch genügend Platz für ein weiteres Zeichen enthalten.

STRING

```
DEFINITION STRING;

IMPORT BT := BasicTypes;

TYPE
 STRING = POINTER TO STRINGDesc;
 STRINGDesc = RECORD (BT.COMPAREABLEDesc)
   chars : BT.DynString;
   PROCEDURE (a:STRING) Capacity(): LONGINT;
   PROCEDURE (a:STRING) Compare(
                     b: BT.COMPAREABLE): LONGINT;
   PROCEDURE (a:STRING) Equal(
                     b: BT.COMPAREABLE): BOOLEAN;
   PROCEDURE (a:STRING) Count(): LONGINT;
   PROCEDURE (a:STRING) Adapt(s: ARRAY OF CHAR);
   PROCEDURE (a:STRING) Append(b: STRING);
   PROCEDURE (a:STRING) Clear;
   PROCEDURE (a:STRING) Copy(b: STRING);
   PROCEDURE (a:STRING) Empty(): BOOLEAN;
   PROCEDURE (a:STRING) Extend(c: CHAR);
   PROCEDURE (a:STRING) ExtendString(
                     s: ARRAY OF CHAR);
   PROCEDURE (a:STRING) FillBlank;
   PROCEDURE (a:STRING) HashCode(): LONGINT;
   PROCEDURE (a:STRING) Head(n: LONGINT);
   PROCEDURE (a:STRING) IndexOf(c: CHAR;
                     i: LONGINT): LONGINT;
   PROCEDURE (a:STRING) LeftAdjust;
   PROCEDURE (a:STRING) Precede(c: CHAR);
   PROCEDURE (a:STRING) Prepend(b: STRING);
   PROCEDURE (a:STRING) Put(c: CHAR;
                     i: LONGINT);
   PROCEDURE (a:STRING) Remove(i: LONGINT);
   PROCEDURE (a:STRING) RemoveAllOccurences(
                     c: CHAR);
   PROCEDURE (a:STRING) Resize(newsize: LONGINT);
   PROCEDURE (a:STRING) Enlarge(newsize: LONGINT);
   PROCEDURE (a:STRING) RightAdjust;
```

```
    PROCEDURE (a:STRING) Shrink(t: STRING;
                        n1, n2: LONGINT);
    PROCEDURE (a:STRING) Substring(
                        n1, n2: LONGINT): STRING;
    PROCEDURE (a:STRING) Tail(n: LONGINT);
    PROCEDURE (a:STRING) ToC(): SYSTEM.ADDRESS;
    PROCEDURE (a:STRING) ToInteger(): LONGINT;
    PROCEDURE (a:STRING) ToLower;
    PROCEDURE (a:STRING) ToUpper;
  END;

CONST
  Blank = " ";

PROCEDURE Create(n: LONGINT): STRING;
PROCEDURE CreateString(s: ARRAY OF CHAR): STRING;

END STRING.
```

Dieses leistungsstarke Oberon-2 Modul stellt einen Recordtyp *STRING* zur Verfügung, in dem Zeichenketten beliebiger Länge gespeichert werden können. Dabei wird die Länge der Zeichenketten automatisch den jeweiligen Anforderungen angepaßt. Das Modul ist stark an die in [Meyer 92] definierte STRING-Klasse angelehnt.

Über viele typgebundene Prozeduren kann mit der Zeichenkette gearbeitet werden. *STRING.chars* enthält einen Zeiger auf ein offenes Feld über das direkt auf die einzelnen Zeichen zugegriffen werden kann. Alle anderen Zugriffe müssen über Aufrufe von (typgebundenen) Prozeduren geschehen:

PROCEDURE Create(n: LONGINT): STRING;

> Es wird ein neues *STRING*-Objekt mit mindestens der Kapazität *n* erzeugt, das heißt es wird Speicher für mindestens *n* Zeichen reserviert. Die erzeugte Zeichenkette ist leer.

PROCEDURE CreateString(s: ARRAY OF CHAR): STRING;

Es wird ein neues *STRING*-Objekt mit einer Kapazität, die größer als *LEN(s)* ist, erzeugt. Ihm wird die Zeichenkette *s* zugewiesen.

PROCEDURE (a: STRING) Capacity(): LONGINT;

Das Ergebnis ist die Kapazität von *a*.

PROCEDURE (a: STRING) Compare(
b: BT.COMPAREABLE): LONGINT;

Diese von *BasicTypes.COMPAREABLE* geerbte und redefinierte Prozedur vergleicht die beiden Zeichenketten und ergibt je nach dem ob *a* kleiner als *b*, *a* gleich *b* oder *a* größer als *b* ist einen Wert kleiner als *0*, gleich *0* oder größer als *0* zurück.

PROCEDURE (a: STRING) Equal(
b: BT.COMPAREABLE): BOOLEAN;

Das Ergebnis ist genau dann wahr, wenn die beiden Zeichenketten die gleiche Länge haben und zeichenweise übereinstimmen.

Less, LessOrEqual, Higher, HigherOrEqual

Diese von *BasicTypes.COMPAREABLE* geerbten Prozeduren können auch auf Objekte des Typs *STRING* angewendet werden.

PROCEDURE (a: STRING) Count(): LONGINT;

Ergebnis ist die Länge von *a*.

PROCEDURE (a: STRING) Adapt(s: ARRAY OF CHAR);

Die Zeichenkette *s* wird in *a* kopiert.

PROCEDURE (a: STRING) Append(b: STRING);

Die Zeichenkette *b* wird hinten an *a* angehängt.

PROCEDURE (a: STRING) Clear;

Die Zeichenkette *a* wird gelöscht. Danach ist *a.Empty()*.

PROCEDURE (a: STRING) Copy(b: STRING);

Die Zeichenkette *b* wird in *a* kopiert.

PROCEDURE (a: STRING) Empty(): BOOLEAN;

Ergebnis ist *TRUE* wenn *a* leer ist.

PROCEDURE (a: STRING) Extend(c: CHAR);
PROCEDURE (a: STRING) ExtendString(
s: ARRAY OF CHAR);

Das Zeichen *c* bzw. die Zeichenkette *s* wird hinten an *a* angehängt.

PROCEDURE (a: STRING) FillBlank;

Alle Zeichen von *a*, also die Zeichen *0* bis *a.Capacity()-1*, werden mit dem Leerzeichen *Blank* (" ") gefüllt.

PROCEDURE (a: STRING) HashCode(): LONGINT;

Es wird eine Positive *LONGINT*-Zahl aus *a* berechnet, der zum Speichern von *a* in einer Hash-Tabelle (siehe [Wirth 83]) benutzt werden kann.

PROCEDURE (a: STRING) Head(n: LONGINT);

Alle Zeichen aus *a* ab dem Zeichen an Position *n* werden gelöscht. Übrig bleiben die ersten *n* Zeichen von *a*, der 'Kopf' von *a*.

PROCEDURE (a: STRING) IndexOf(c: CHAR;
i: LONGINT): LONGINT;

a wird ab der Position *i* nach dem nächsten Vorkommen des Zeichens *c* untersucht. Wird *c* gefunden, so wird der gefundene Index zurückgegeben. Sonst ist das Ergbebnis *-1*.

PROCEDURE (a: STRING) LeftAdjust;

Alle führenden Leerzeichen aus *a* werden entfernt.

PROCEDURE (a: STRING) Precede(c: CHAR);

Das Zeichen *c* wird als neues erstes Zeichen in die Zeichenkette eingefügt. Alle anderen Zeichen werden um eine Position nach hinten geschoben.

PROCEDURE (a: STRING) Prepend(b: STRING);

Die Zeichenkette *b* wird vorne in *a* eingefügt, die Zeichen in *a* werden dazu um die Länge von *b* nach hinten verschoben.

PROCEDURE (a: STRING) Put(c: CHAR; i: LONGINT);

Das Zeichen an der Position *i* in *a* wird durch *c* ersetzt.

PROCEDURE (a: STRING) Remove(i: LONGINT);

Das Zeichen an der Position *i* wird aus *a* entfernt. Alle Zeichen hinter diesem wandern dabei um eine Position nach links.

PROCEDURE (a: STRING) RemoveAllOccurences(c: CHAR);

Alle Vorkommen des Zeichens *c* werden aus *a* gelöscht. Die Zeichen rechts der gelöschten Zeichen rücken dabei entsprechend nach links.

PROCEDURE (a: STRING) Resize(newsize: LONGINT);
PROCEDURE (a: STRING) Enlarge(newsize: LONGINT);

Diese Prozeduren erhöhen die Kapazität von *a*, so daß *a* Platz für mindestens *newsize* Zeichen hat. *Enlarge* fordert dabei evtl. gleich mehr Zeichen an als erforderlich, so daß bei vielen Aufrufen von *Enlarge*, bei denen die Kapazität jeweils nur geringfügig erhöht würde, nur wenige Speicheranforderungen nötig sind.

PROCEDURE (a: STRING) RightAdjust;

Alle Leerzeichen am Ende von *a* werden gelöscht.

PROCEDURE (a: STRING) Shrink(t: STRING;
n1, n2: LONGINT);

Die Teil-Zeichenkette von der Position *n1* bis zur Position *n2* wird aus *t* nach *a* kopiert.

PROCEDURE (a: STRING) Substring(
n1, n2: LONGINT): STRING;

Es wird ein neues *STRING*-Objekt erzeugt, das alle Zeichen von Position *n1* bis Position *n2* aus *a* enthält.

PROCEDURE (a: STRING) Tail(n: LONGINT);

Bis auf die letzten *n* Zeichen werden alle führenden Zeichen aus *a* gelöscht. *a* enthält danach den 'Schwanz' der Zeichenkette mit der Länge *n*.

PROCEDURE (a: STRING) ToC(): SYSTEM.ADDRESS;

Es wird die Adresse der Zeichenkette zurückgeliefert, wie er von C-Routinen und dem Amiga-Betriebssystem oft benötigt wird. Für Oberon-Programme ist dieser Wert nutzlos.

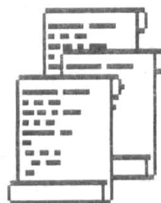
PROCEDURE (a: STRING) ToInteger(): LONGINT;

Für diese Funktion muß *a* eine Dezimalzahl, evtl. mit Vorzeichen, enthalten. Das Ergebnis ist der Integerwert dieser Zahl.

PROCEDURE (a: STRING) ToLower;
PROCEDURE (a: STRING) ToUpper;

Alle Zeichen der Zeichenkette werden in Klein- (*ToLower*) bzw. in Großbuchstaben (*ToUpper*) umgewandelt.

21. Modulbibliothek: Mathematikmodule



Die im folgenden beschriebenen Module bieten verschiedene Arten von Zahlen an, wie etwa komplexe Zahlen oder beliebig große natürliche Zahlen. Zudem gibt es Module die speziellen mathematischen Funktionen exportieren.

BigIntegers

```
DEFINITION BigIntegers;

IMPORT BT := BasicTypes;

TYPE
  BigInteger = POINTER TO BigIntegerDesc;
  BigIntegerDesc = RECORD (BT.RINGDesc)
    negative : BOOLEAN;
    digits : POINTER TO ARRAY OF INTEGER;
    PROCEDURE (a:BigInteger) Compare(
                   b: BT.COMPAREABLE): LONGINT;
    PROCEDURE (m:BigInteger) Add(
                   n: BT.GROUP): BT.GROUP;
    PROCEDURE (m:BigInteger) Neg(): BT.GROUP;
    PROCEDURE (m:BigInteger) Norm(): LONGREAL;
    PROCEDURE (m:BigInteger) Mul(
                   n: BT.RING): BT.RING;
    PROCEDURE (m:BigInteger) ConvertToString
                    (): BT.DynString;
  END;

CONST
  MaxDigit = 1000;

PROCEDURE Create(init: LONGINT): BigInteger;

END BigIntegers.
```

Der hier definierte Recordtyp dient zum Speichern und Rechnen mit beliebig großen ganzen Zahlen. Die Größe der Zahlen wird nur durch den Systemspeicher begrenzt. Da die Menge **Z** der ganzen Zahlen einen Ring bezüglich der Addition und der Multiplikation bilden, sind sie als Erweiterungen von *BasicTypes.RING* definiert. Über die zum Lesen exportierten Recordelemente *negative* und *digits* kann direkt auf die Zahl zugegriffen werden. Dabei zeigt *negative* das Vorzeichen der Zahl an. Das Feld *digits^* enthält die Zahl selbst. Dabei ist *0<=digits[i]<MaxDigit* für alle *i* aus *0..LEN(digits^)-1*. Der Betrag der dargestellten Zahl ist $\sum_{i \in 0..LEN(digits^\wedge)-1} MaxDigit^{i} * digits[i]$.

Auf die Zahlen kann mit folgenden Prozeduren zugegriffen werden:

PROCEDURE Create(init: LONGINT): BigInteger;

Es wird ein *BigInteger*-Objekt erzeugt. Es wird mit der Zahl *init* vorbelegt.

PROCEDURE (a: BigInteger) Compare(
b: BT.COMPAREABLE): LONGINT;

Diese von *BasicTypes.COMPAREABLE* geerbte und redefinierte Prozedur vergleicht die beiden Zahlen *a* und *b* und ergibt, je nach dem, ob *a* kleiner als, gleich oder größer als *b* ist, einen Wert kleiner, gleich oder größer als *0*.

Equal, Less, LessOrEqual, Higher, HigherOrEqual

Diese von *BasicTypes.COMPAREABLE* geerbten Prozeduren können auch für Objekte des Typs *BigInteger* angewendet werden.

PROCEDURE (m: BigInteger) Add(n: BT.GROUP): BT.GROUP;

Es wird ein neues *BigInteger*-Objekt erzeugt dem die Summe der beiden *BigIntegers m* und *n* zugewiesen wird.

PROCEDURE (m: BigInteger) Neg(): BT.GROUP;

Es wird ein neues *BigInteger*-Objekt erzeugt mit gleichem Betrag wie *m*, aber anderem Vorzeichen.

PROCEDURE (m: BigInteger) Norm(): LONGREAL;

Ergebnis ist der Betrag von *m*.

PROCEDURE (m: BigInteger) Mul(n: BT.RING): BT.RING;

Es wird ein neues *BigInteger*-Objekt erzeugt, dem das Produkt der beiden *BigIntegers* *m* und *n* zugewiesen wird.

PROCEDURE (m: BigInteger) ConvertToString(): BT.DynString;

Die Zahl *m* wird in eine Dezimalzahl umgewandelt, die als Zeichenkette zurückgegeben wird.

Das folgende Beispielprogramm berechnet alle Fakultäten von eins bis 100. Damit der Quelltext anschaulicher wird macht das Programm von den Spracherweiterungen von Amiga Oberon gebrauch:

```
MODULE Fak;
IMPORT BI := BigIntegers, Out;
VAR
  i: INTEGER;
  bi: BI.BigInteger;
BEGIN
  bi := BI.Create(1);
  FOR i:=1 TO 100 DO
    Out.Int(i,4); Out.String("! =");
    bi := bi.Mul(BI.Create(i))(BI.BigInteger);
    Out.String(bi.ConvertToString()^);
    Out.Ln;
  END;
END Fak.
```

BigQuotients

```
DEFINITION BigQuotients;

IMPORT BT := BasicTypes, BI := BigIntegers;

TYPE
  BigQuotient = POINTER TO BigQuotientDesc;
  BigQuotientDesc = RECORD (BT.FIELDDesc)
    p,q : BI.BigInteger;
    PROCEDURE (a:BigQuotient) Compare(
                         b: BT.COMPAREABLE): LONGINT;
    PROCEDURE (m:BigQuotient) Add(
                         n: BT.GROUP): BT.GROUP;
    PROCEDURE (m:BigQuotient) Neg(): BT.GROUP;
    PROCEDURE (m:BigQuotient) Norm(): LONGREAL;
    PROCEDURE (m:BigQuotient) Mul(
                         n: BT.RING): BT.RING;
    PROCEDURE (m:BigQuotient) Inv(): BT.FIELD;
    PROCEDURE (m:BigQuotient) InvAllowed
                          (): BOOLEAN;
    PROCEDURE (n:BigQuotient) ConvertToString
                          (): BT.DynString;
  END;

PROCEDURE Create(p, q: LONGINT): BigQuotient;

END BigQuotients.
```

Dieses Modul definiert einen Typ für exakte Bruchzahlen. Da die Menge **Q** der Bruchzahlen einen mathematischen Körper bilden, ist der Typ der Bruchzahlen, *BigQuotient*, hier als Erweiterung von *BasicTypes.FIELD* definiert.

Eine Bruchzahl besteht aus zwei ganzen Zahlen: dem Zähler *p* und dem Nenner *q*. Auf diese Recordelemente vom Typ *BigInteger* kann lesend direkt zugegriffen werden. BigQuotients bietet die folgenden Prozeduren:

PROCEDURE Create(p, q: LONGINT): BigQuotient;

Es wird ein neuer Bruch erzeugt. Ihm wird der Wert *p/q* zugewiesen.

**PROCEDURE (a: BigQuotient) Compare(
b: BT.COMPAREABLE): LONGINT;**

Diese von *BasicTypes.COMPAREABLE* geerbte und redefinierte Prozedur vergleicht die beiden Brüche *a* und *b* und ergibt, je nach dem ob *a* kleiner als, gleich oder größer als *b* ist einen Wert kleiner, gleich oder größer als *0*.

Equal, Less, LessOrEqual, Higher, HigherOrEqual

Diese von *BasicTypes.COMPAREABLE* geerbten Prozeduren können auch für Objekte des Typs *BigInteger* angewendet werden.

**PROCEDURE (m: BigQuotient) Add(
n: BT.GROUP): BT.GROUP;**

Es wird ein neues *BigQuotient*-Objekt erzeugt dem die Summe von *m* und *n* zugewiesen wird.

PROCEDURE (m: BigQuotient) Neg(): BT.GROUP;

Es wird ein neues *BigQuotient*-Objekt erzeugt. Ihm wird *-m* zugewiesen.

PROCEDURE (m: BigQuotient) Norm(): LONGREAL;

Das Ergebnis ist der Betrag von *m*.

PROCEDURE (m: BigQuotient) Mul(n: BT.RING): BT.RING;

Es wird ein neues *BigQuotient*-Objekt erzeugt dem das Produkt von *m* und *n* zugewiesen wird.

PROCEDURE (m: BigQuotient) Inv(): BT.FIELD;

Es wird ein neues *BigQuotient*-Objekt erzeugt, dem der Kehrwert von *m* zugewiesen wird.

PROCEDURE (m: BigQuotient) InvAllowed(): BOOLEAN;

Das Ergebnis ist *TRUE* wenn *m* nicht null ist.

PROCEDURE (n: BigQuotient) ConvertToString (): BT.DynString;

Der Bruch *m* wird in eine Zeichenkette umgewandelt, die aus zwei Dezimalzahlen getrennt von einem Schrägstrich besteht.

COMPLEX

```
DEFINITION COMPLEX;

IMPORT BT := BasicTypes;

TYPE
  COMPLEX = POINTER TO COMPLEXDesc;
  COMPLEXDesc = RECORD (BT.FIELDDesc)
    re,im : LONGREAL;
    PROCEDURE (m:COMPLEX) Add(
                    n: BT.GROUP): BT.GROUP;
    PROCEDURE (m:COMPLEX) Neg(): BT.GROUP;
    PROCEDURE (m:COMPLEX) Sub(
                    n: BT.GROUP): BT.GROUP;
    PROCEDURE (m:COMPLEX) Mul(n: BT.RING): BT.RING;
    PROCEDURE (m:COMPLEX) InvAllowed(): BOOLEAN;
```

```
    PROCEDURE (m:COMPLEX) Inv(): BT.FIELD;
    PROCEDURE (m:COMPLEX) Div(
                     n: BT.FIELD): BT.FIELD;
    PROCEDURE (m:COMPLEX) isEqual(
                     n: BT.COMPAREABLE): BOOLEAN;
    PROCEDURE (m:COMPLEX) Compare(
                     n: BT.COMPAREABLE): LONGINT;
    PROCEDURE (m:COMPLEX) Norm(): LONGREAL;
    PROCEDURE (m:COMPLEX) Argument(): LONGREAL;
  END;

PROCEDURE Create(re, im: LONGREAL): COMPLEX;

END COMPLEX.
```

Der Körper **C** der komplexen Zahlen wird von den Typen dieses Moduls nachgebildet. Der Typ *COMPLEX* ist daher als eine Erweiterung von *BasicTypes.FIELD* definiert.

Über die Recordelement *re* und *im* kann direkt der Real- und Imaginärteil der komplexen Zahl bestimmt werden. Zum Arbeiten mit komplexen Zahlen stehen folgende Prozeduren bereit:

PROCEDURE Create(re, im: LONGREAL): COMPLEX;

Es wird eine neue Komplexe Zahl mit dem Realteil *re* und dem Imaginärteil *im* erzeugt.

PROCEDURE (m: COMPLEX) Add(
n: BT.GROUP): BT.GROUP;

Es wird eine neue komplexe Zahl erzeugt. Ihr wird der Summe von *m* und *n* zugewiesen.

PROCEDURE (m: COMPLEX) Neg(): BT.GROUP;

Es wird eine neue komplexe Zahl erzeugt. Ihr wird der Wert *-m* zugewiesen.

PROCEDURE (m: COMPLEX) Sub(
n: BT.GROUP): BT.GROUP;

Es wird eine neue komplexe Zahl erzeugt. Ihr wird der Wert *m-n* zugewiesen.

PROCEDURE (m: COMPLEX) Mul(n: BT.RING): BT.RING;

Es wird eine neue komplexe Zahl erzeugt. Ihr wird der Wert *m*n* zugewiesen.

PROCEDURE (m: COMPLEX) Inv(): BT.FIELD;

Es wird eine neue komplexe Zahl erzeugt. Ihr wird der Kehrwert von *m* zugewiesen.

PROCEDURE (m: COMPLEX) InvAllowed(): BOOLEAN;

Das Ergebnis ist *TRUE*, wenn *m* nicht null ist.

PROCEDURE (m: COMPLEX) Div(n: BT.FIELD): BT.FIELD;

Es wird eine neue komplexe Zahl erzeugt. Ihr wird der Wert von *m/n* zugewiesen.

PROCEDURE (m: COMPLEX) Norm(): LONGREAL;

Das Ergebnis ist der Betrag |*m*|.

PROCEDURE (m: COMPLEX) Argument(): LONGREAL;

m.Argument() bestimmt das Argument von *m*. Dies ist der Winkel zur x-Achse bei der Darstellung von *m* in Polarkoordinaten.

PROCEDURE (m: COMPLEX) isEqual(
n: BT.COMPAREABLE): BOOLEAN;

Das Ergebnis ist *TRUE* wenn *m* und *n* den gleichen Wert enthalten.

PROCEDURE (m: COMPLEX) Compare(
n: BT.COMPAREABLE): LONGINT;

Diese von *BasicTypes.COMPAREABLE* geerbte und redefinierte Prozedur vergleicht die beiden Beträge von *a* und *b* und ergibt, je nach dem ob |*a*| kleiner als, gleich oder größer als |*b*| ist einen Wert kleiner, gleich oder größer als *0*.

Equal, Less, LessOrEqual, Higher, HigherOrEqual

Diese von *BasicTypes.COMPAREABLE* geerbten Prozeduren können auch für Objekte des Typs *COMPLEX* angewendet werden. Achtung: *a.Equal(b)* vergleicht nicht die Zahlen, sondern nur ihre Beträge! Zum Vergleichen von komplexen Zahlen muß *isEqual* verwendet werden.

Da die Menge der komplexen Zahlen nicht geordnet ist, sind diese Zahlen nicht direkt vergleichbar werden, mit ihren Beträgen ist dies jedoch möglich.

MATHLIB

```
DEFINITION MATHLIB;

PROCEDURE ACOS  (x: LONGREAL): LONGREAL;
PROCEDURE ASIN  (x: LONGREAL): LONGREAL;
PROCEDURE ATAN  (x: LONGREAL): LONGREAL;
PROCEDURE ATANH (x: LONGREAL): LONGREAL;
PROCEDURE COS   (x: LONGREAL): LONGREAL;
```

```
PROCEDURE COSH   (x: LONGREAL): LONGREAL;
PROCEDURE ETOX   (x: LONGREAL): LONGREAL;
PROCEDURE LOG10  (x: LONGREAL): LONGREAL;
PROCEDURE LOG2   (x: LONGREAL): LONGREAL;
PROCEDURE LOGN   (x: LONGREAL): LONGREAL;
PROCEDURE SIN    (x: LONGREAL): LONGREAL;
PROCEDURE SINH   (x: LONGREAL): LONGREAL;
PROCEDURE SQR    (x: LONGREAL): LONGREAL;
PROCEDURE SQRT   (x: LONGREAL): LONGREAL;
PROCEDURE TAN    (x: LONGREAL): LONGREAL;
PROCEDURE TANH   (x: LONGREAL): LONGREAL;
PROCEDURE TENTOX(x: LONGREAL): LONGREAL;
PROCEDURE TWOTOX(x: LONGREAL): LONGREAL;
PROCEDURE INT    (x: LONGREAL): LONGREAL;

END MATHLIB.
```

Dieses Modul entspricht dem compilerinternen Modul *MATHLIB*, das bei der Codeerzeugung für einen Fließkommaprozessor sichtbar ist. Damit Module, die mit *MATHLIB* geschrieben wurden, auch ohne den Fließkommaprozessor zu benutzen übersetzt werden können, existiert dieses 'echte' Modul *MATHLIB*. Die Prozeduren entsprechen denen des Compilerinternen Moduls, sie benutzen jedoch die Routinen des AmigaOS um die Berechnungen anzustellen. Eine Beschreibung der Prozeduren befindet sich in Kapitel 14.

Random

```
DEFINITION Random;

PROCEDURE RND(n: INTEGER): INTEGER;
PROCEDURE PutSeed(seed: LONGINT);

END Random.
```

Mit den Prozeduren dieses Moduls können einfach pseudo-Zufallszahlen erzeugt werden:

PROCEDURE RND(n: INTEGER): INTEGER;

RND(n) ergibt eine pseudo-Zuffallszahl zwischen null und *n-1*. Um damit einen Würfel zu simulieren, berechnet man die Augenzahl mit *RND(6)+1*.

PROCEDURE PutSeed(seed: LONGINT);

Der Anfangswert des Zufallsgenerators wird mit dieser Prozedur gesetzt. Zu Beginn wird dieser mit der Systemzeit initialisiert, so daß gewöhnlich immer unterschiedliche Zufallsfolgen erzeugt werden. Benötigt man jedoch mehrmals dieselbe Zahlenfolge, so kann der Anfangswert mit *PutSeed* auf denselben Wert gesetzt werden.

VECTOR

```
DEFINITION VECTOR;

IMPORT BT := BasicTypes;

TYPE
  VECTOR = POINTER TO VECTORDesc;
  VECTORDesc = RECORD (BT.GROUPDesc)
    PROCEDURE (v:VECTOR) Get(n: LONGINT): LONGREAL;
    PROCEDURE (v:VECTOR) Put(n: LONGINT;
                         x: LONGREAL);
    PROCEDURE (v:VECTOR) Dimension(): LONGINT;
    PROCEDURE (v:VECTOR) Add(
                         n: BT.GROUP): BT.GROUP;
    PROCEDURE (v:VECTOR) Neg(): BT.GROUP;
    PROCEDURE (v:VECTOR) Sub(
                         n: BT.GROUP): BT.GROUP;
    PROCEDURE (v:VECTOR) Norm(): LONGREAL;
    PROCEDURE (v:VECTOR) Mul3(n: VECTOR): VECTOR;
    PROCEDURE (v:VECTOR) isEqual(
                         n: BT.COMPAREABLE): BOOLEAN;
```

```
    PROCEDURE (a:VECTOR) Compare(
                         b: BT.COMPAREABLE): LONGINT;
    PROCEDURE (a:VECTOR) ScalarProduct(
                         b: VECTOR): LONGREAL;
    PROCEDURE (a:VECTOR) Angle(
                         b: VECTOR): LONGREAL;
    PROCEDURE (a:VECTOR) SMul(r: LONGREAL): VECTOR;
  END;

PROCEDURE Create(dim: LONGINT): VECTOR;

END VECTOR.
```

Dieses Modul stellt Typen für die Elemente der Vektorräume $\mathbf{R}^n$ zur Verfügung. Da Vektorräume mit der Vektoraddition eine Gruppe bilden, ist *VECTORDesc* als Erweiterung von *BasicTypes.GROUPDesc* definiert.

Die Prozeduren aus *VECTOR*:

PROCEDURE Create(dim: LONGINT): VECTOR;

Es wird ein neues *VECTOR*-Objekt aus $\mathbf{R}^n$ erzeugt. Ihm wird der Nullvektor zugewiesen.

PROCEDURE (v: VECTOR) Get(n: LONGINT): LONGREAL;

Die Koordinate in der Dimension *n* des Vektors wird zurückgegeben. *n* muß dabei zwischen *0* und *v.Dimension()-1* liegen.

PROCEDURE (v: VECTOR) Put(n: LONGINT;
x: LONGREAL);

Die Koordinate in der Dimension *n* des Vektors wird auf *x* gesetzt. *n* muß dabei zwischen *0* und *v.Dimension()-1* liegen.

PROCEDURE (v: VECTOR) Dimension(): LONGINT;

Das Ergebnis ist die Dimension des Vektorraums, derssen Werte *v* annehmen kann. Dies ist der Wert, der bei *Create* übergeben wurde.

PROCEDURE (v: VECTOR) Add(n: BT.GROUP): BT.GROUP;
PROCEDURE (v: VECTOR) Neg(): BT.GROUP;
PROCEDURE (v: VECTOR) Sub(n: BT.GROUP): BT.GROUP;

Es wird jeweils ein neues *VECTOR*-Objekt der Dimension von *v* erzeugt und mit dem Wert *v+n* (*Add*), *-v* (*Neg*) bzw. *v-n* (*Sub*) belegt. *n* muß dabei die gleiche Dimension wie *v* haben.

PROCEDURE (v: VECTOR) Norm(): LONGREAL;

Das Ergebnis ist der Betrag des Vektors *v*.

PROCEDURE (v: VECTOR) Mul3(n: VECTOR): VECTOR;

v.Mul(n) alloziert ein neues, dreidimensionales *VECTOR*-Objekt und weist ihm den Wert des Vektorprodukts des R3 zu. *v* und *n* müssen dabei die Dimension *3* haben.

PROCEDURE (v: VECTOR) isEqual(
n: BT.COMPAREABLE): BOOLEAN;

Das Ergebnis ist *TRUE* wenn *v* und *n* den gleichen Vektor enthalten. *v* und *n* müssen die gleiche Dimension haben.

PROCEDURE (a: VECTOR) Compare(
b: BT.COMPAREABLE): LONGINT;

Diese von *BasicTypes.COMPAREABLE* geerbte und redefinierte Prozedur vergleicht die beiden Beträge von *a* und *b* und ergibt, je nach dem ob |*a*| kleiner als, gleich oder größer als |*b*| ist einen

Wert kleiner, gleich oder größer als *0*. Da ein Vektorraum $\mathbf{R}^n$ für *n>0* nicht geordnet ist, können nicht die Vektoren selbst, sondern lediglich ihre Beträge verglichen werden.

Equal, Less, LessOrEqual, Higher, HigherOrEqual

Diese von *BasicTypes.COMPAREABLE* geerbten Prozeduren können auch für Objekte des Typs *COMPLEX* angewendet werden. Achtung: *a.Equal(b)* vergleicht nicht die Vektoren, sondern nur ihre Beträge! Zum Vergleichen von Vektoren muß *isEqual* verwendet werden.

**PROCEDURE (a: VECTOR) ScalarProduct(
b: VECTOR): LONGREAL;**

Das Ergebnis ist das Skalarprodukt von *a* und *b*. Die beiden Vektoren müssen die gleiche Dimension haben.

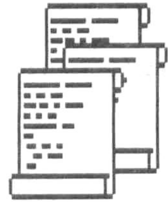
PROCEDURE (a: VECTOR) Angle(b: VECTOR): LONGREAL;

Das Ergebnis ist der Winkel zwischen den Vektoren *a* und *b*.

PROCEDURE (a: VECTOR) SMul(r: LONGREAL): VECTOR;

Der Vektor *a* wird mit dem Skalar *r* multipliziert. Das Ergebnis ist ein neuer Vektor mit der gleichen Richtung von *a*, der jedoch um den Faktor *a* gestreckt wurde.

22. Modulbibliothek: Ein- und Ausgabe



Eine grundlegende Eigenschaft von sinnvollen Computerprogrammen ist es, daß sie eine Eingabe erhalten, diese bearbeiten und danach ein Resultat ausgeben. In diesem Kapitel werden Module beschrieben, die Prozeduren zur Ein- und Ausgabe von Daten bieten.

Arguments

```
DEFINITION Arguments;

IMPORT d := Dos;

VAR
  oldCurrentDir : d.FileLockPtr;

PROCEDURE NumArgs(): INTEGER;
PROCEDURE GetArg(arg: INTEGER;
                 VAR argument: ARRAY  OF CHAR);
PROCEDURE GetLock(num: INTEGER): d.FileLockPtr;

END Arguments.
```

Beim Start von Programmen von der Shell aus können in der Kommandozeile Argumente an das Programm übergeben werden. Auch beim Start von der Workbench können über erweitertes Anwählen mit der Umschalttaste (Shift) eine Reihe an Dateien an das gestartete Programm übergeben werden.

Damit Oberon-Programme nicht immer berücksichtigen müssen, ob sie von einer Shell oder von der Workbench aufgerufen wurden und die verschiedenen Arten von Argumenten auswerten müssen, bietet *Arguments* eine einheitliche Zugriffsweise sowohl auf die Shell- als auch auf die Workbench-Argumente.

Die Prozeduren von *Arguments*:

PROCEDURE NumArgs(): INTEGER;

Das Ergebnis dieser Prozedur ist die Anzahl der Argumente, die dem Programm übergeben wurden. Beim Workbench-Start ist dies die Anzahl der erweitert angewählten Piktogramme. Beim Shell-Start ist dies die Zahl der durch Leerzeichen getrennten Argumente. Die einzelnen Argumente können hier mit Anführungszeichen umschlossen sein.

PROCEDURE GetArg(arg: INTEGER;
VAR argument: ARRAY OF CHAR);

Liegt *arg* zwischen eins und *NumArgs()* so wird das Argument mit der Nummer *arg* nach *argument* kopiert. Ist *arg* null wird der Name, unter dem das Programm gestartet wurde, nach *argument* kopiert.

Wurde das Programm von der Workbench gestartet so ändert *GetArg* auch das aktuelle Verzeichnis auf das, in dem sich das ausgelesene Argument befindet, so daß die Datei gleich geöffnet werden kann.

Damit das aktuelle Verzeichnis leicht wieder auf das beim Programmstart gesetzt werden kann, wird die Variable *oldCurrentDir* exportiert, die einen *FileLockPtr* auf dieses Verzeichnis enthält.

PROCEDURE GetLock(num: INTEGER): d.FileLockPtr;

Das Ergebnis ist ein *FileLockPtr* des Verzeichnisses, in dem sich das Argument *num* befindet. Nach einem Shell-Start ist dies immer das beim Start aktuelle Verzeichnis.

Display

```
DEFINITION Display;

IMPORT
  li := Lists,
  I := Intuition,
  u := Utility,
  g := Graphics,
  e := Exec;

TYPE
  DispEl = RECORD (li.Node)
    title : e.STRPTR;
    rp : g.RastPortPtr;
    font : g.TextFontPtr;
    width : INTEGER;
    height : INTEGER;
    turtleX : REAL;
    turtleY : REAL;
    turtleDir : REAL;
    pen : BOOLEAN;
    cursor : BOOLEAN;
    curX : INTEGER;
    curY : INTEGER;
    curXAbs : INTEGER;
    curYAbs : INTEGER;
    txtWidth : INTEGER;
    txtHeight : INTEGER;
    gzz : BOOLEAN;
    left,top : INTEGER;
  END;
  Screen = RECORD (DispEl)
    screen : I.ScreenPtr;
    li : g.LayerInfoPtr;
    layer : g.LayerPtr;
  END;
  Window = RECORD (DispEl)
    window : I.WindowPtr;
  END;
```

```
  DispElPtr = POINTER TO DispEl;
  ScreenPtr = POINTER TO Screen;
  WindowPtr = POINTER TO Window;

CONST
  line = -1;
  dots = 5555H;
  bigdots = 3333H;
  broken = 0F0FH;

PROCEDURE OpenScreen(scrn: ScreenPtr;
                title: ARRAY OF CHAR;
                x, y, w, h: INTEGER;
                d: SHORTINT;
                hires, lace: BOOLEAN): BOOLEAN;
PROCEDURE OpenWindow(win: WindowPtr;
                title: ARRAY OF CHAR;
                x, y, w, h: INTEGER;
                screen: I.ScreenPtr): BOOLEAN;
PROCEDURE OpenWindowTags(win: WindowPtr;
                gadg: I.GadgetPtr;
                gzz: BOOLEAN;
                title: ARRAY OF CHAR;
                x, y, w, h: INTEGER;
                screen: I.ScreenPtr;
                activate: BOOLEAN;
                tags: sys.ADDRESS): BOOLEAN;
PROCEDURE OpenWindowX(win: WindowPtr;
                gadg: I.GadgetPtr;
                gzz: BOOLEAN;
                title: ARRAY OF CHAR;
                x, y, w, h: INTEGER;
                activate: BOOLEAN;
                screen: I.ScreenPtr): BOOLEAN;
PROCEDURE Close(d: DispElPtr);
PROCEDURE Init(d: DispElPtr);
PROCEDURE SetColors(s: ScreenPtr;
                VAR cols: ARRAY OF INTEGER);
PROCEDURE SetCol(s: ScreenPtr;
                num, R, G, B: INTEGER);
```

```
PROCEDURE NumToRGB(num: INTEGER;
               VAR r, g, b: INTEGER);
PROCEDURE RGBToNum(r, g, b: INTEGER): INTEGER;
PROCEDURE FrontPen(d: DispElPtr; pen: SHORTINT);
PROCEDURE BackPen(d: DispElPtr; pen: SHORTINT);
PROCEDURE Jam1(d: DispElPtr);
PROCEDURE Jam2(d: DispElPtr);
PROCEDURE Complement(d: DispElPtr);
PROCEDURE LinePattern(d: DispElPtr; pat: INTEGER);
PROCEDURE Clear(d: DispElPtr);
PROCEDURE Line(d: DispElPtr;
               x1, y1, x2, y2: INTEGER);
PROCEDURE Dot(d: DispElPtr; x, y: INTEGER);
PROCEDURE DotColor(d: DispElPtr;
               x, y: INTEGER): INTEGER;
PROCEDURE Rect(d: DispElPtr; x, y, w, h: INTEGER);
PROCEDURE Box(d: DispElPtr; x, y, w, h: INTEGER);
PROCEDURE Move(d: DispElPtr; x, y: INTEGER);
PROCEDURE Draw(d: DispElPtr; x, y: INTEGER);
PROCEDURE Text(d: DispElPtr; x, y: INTEGER;
               s: ARRAY OF CHAR);
PROCEDURE Circle(d: DispElPtr; x, y, r: INTEGER);
PROCEDURE Ellipse(d: DispElPtr;
               x, y, rx, ry: INTEGER);
PROCEDURE Font(d: DispElPtr;
               name: ARRAY OF CHAR;
               height: INTEGER): BOOLEAN;
PROCEDURE SetTurtlePos(d: DispElPtr; x, y: REAL);
PROCEDURE GetTurtlePos(d: DispElPtr;
               VAR x, y: REAL);
PROCEDURE SetTurtleDir(d: DispElPtr; dir: REAL);
PROCEDURE GetTurtleDir(d: DispElPtr): REAL;
PROCEDURE SetPen(d: DispElPtr);
PROCEDURE LiftPen(d: DispElPtr);
PROCEDURE Forward(d: DispElPtr; s: REAL);
PROCEDURE TurnLeft(d: DispElPtr; alpha: REAL);
PROCEDURE TurnRight(d: DispElPtr; alpha: REAL);
PROCEDURE SetCursor(d: DispElPtr;
               on: BOOLEAN): BOOLEAN;
PROCEDURE CursorOn(d: DispElPtr);
```

```
PROCEDURE CursorOff(d: DispElPtr);
PROCEDURE Position(d: DispElPtr; x, y: INTEGER);
PROCEDURE Plain(d: DispElPtr);
PROCEDURE UnderLinedOn(d: DispElPtr);
PROCEDURE UnderLinedOff(d: DispElPtr);
PROCEDURE BoldOn(d: DispElPtr);
PROCEDURE BoldOff(d: DispElPtr);
PROCEDURE ItalicOn(d: DispElPtr);
PROCEDURE ItalicOff(d: DispElPtr);
PROCEDURE Home(d: DispElPtr);
PROCEDURE ClrHome(d: DispElPtr);
PROCEDURE ScrollUp(d: DispElPtr);
PROCEDURE ScrollUpN(d: DispElPtr; n: INTEGER);
PROCEDURE ScrollDown(d: DispElPtr);
PROCEDURE ScrollDownN(d: DispElPtr; n: INTEGER);
PROCEDURE InsertLine(d: DispElPtr; n: INTEGER);
PROCEDURE DeleteLine(d: DispElPtr; n: INTEGER);
PROCEDURE WriteLn(d: DispElPtr);
PROCEDURE Write(d: DispElPtr; Str: ARRAY OF CHAR);

END Display.
```

Dieses mächtige Modul beinhaltet eine große Zahl an Zeichenprozeduren, Prozeduren für Turtle-Grafik und Prozeduren zur komfortablen Textausgabe. Dieses Modul wird vom Editor OEd für die Textfenster verwendet.

Fenster und Bildschirme (Screens) werden dabei gleich behandelt. Alle Zeichenprozeduren erhalten als Parameter einen Zeiger auf ein *DispEl*-Record. *Window* und *Screen* sind dabei als Erweiterungen von *DispEl* definiert, so daß den Prozeduren sowohl ein Fenster wie auch ein Bildschirm übergeben werden kann.

Die Prozeduren von *Display* sind im Einzelnen:

**PROCEDURE OpenScreen(scrn: ScreenPtr;
title: ARRAY OF CHAR;
x, y, w, h: INTEGER;
d: SHORTINT;
hires, lace: BOOLEAN): BOOLEAN;**

Es wird ein neuer Bildschirm mit dem Namen *title*, der Position *x*, *y*, der Größe *w*, *h* und der Tiefe *d* geöffnet. *title* kann "" sein. Dann wird ein Bildschirm ohne Titelzeile erzeugt. Die Anzahl der Farben, die auf dem Bildschirm dargestellt werden können, kann mit 2^d bestimmt werden. *hires* und *lace* geben an, ob ein Hires- bzw. ein Interlace-Bildschirm geöffnet werden soll. *scrn* muß beim Aufruf von *OpenScreen* auf ein mit *NEW* angefordertes *Screen*-Record zeigen. Das Ergebnis ist *TRUE*, wenn der Bildschirm geöffnet werden konnte.

**PROCEDURE OpenWindow(win: WindowPtr;
title: ARRAY OF CHAR;
x, y, w, h: INTEGER;
screen: I.ScreenPtr): BOOLEAN;**

OpenWindow öffnet ein Fenster mit dem Namen *title* und der Größe *w*, *h* an der Position *x*, *y*. Soll das Fenster auf einem bestimmten Bildschirm geöffnet werden, so muß der Zeiger auf die Intuition-*Screen*-Struktur als *screen* übergeben werden. Hat man mit *OpenScreen* einen solchen Bildschirm geöffnet, so erhält man einen Zeiger auf diese Struktur über *scrn.screen*.

win muß beim Aufruf von *OpenWindow* auf ein mit *NEW* angefordertes *Window*-Record zeigen. Das Ergebnis ist *TRUE*, wenn das Fenster geöffnet werden konnte.

OpenWindowTags, OpenWindowX

Mit diesen Prozeduren können auch Fenster geöffnet werden. Außer den Parametern von *OpenWindow* können hier jedoch

weitere Werte angegeben werden. Die exakte Bedeutung dieser Parameter wird in [RKM: Libraries 92] beschrieben. *gadg* zeigt auf eine Liste von *Gadget*-Strukturen, die das Fenster benutzen soll. Ist *gzz* gesetzt wird ein *GimmeZeroZero*-Fenster geöffnet. Wird in eine solches Fenster außerhalb des inneren Bereichs gezeichnet, wird der Fensterrahmen nicht zerstört. Mit *activate* kann gewählt werden, ob das Fenster beim Öffnen aktiviert werden soll.

OpenWindowTags erlaubt zudem die Übergabe eines Zeigers *tags* auf eine TagListe. Diese wird unter AmigaOS 2.0 an *Intuition.OpenWindowTags* übergeben.

PROCEDURE Close(d: DispElPtr);

Das Fenster oder der Bildschirm *d* wird geschlossen.

PROCEDURE Init(d: DispElPtr);

Diese Prozedur sollte immer dann aufgerufen werden, wenn die Größe des Fensters *d* verändert wurde, also z.B. wenn an dem *userPort* des Fensters eine *newSize*-Nachricht angekommen ist (siehe [RKM: Libraries 92]).

PROCEDURE SetColors(s: ScreenPtr;
VAR cols: ARRAY OF INTEGER);
PROCEDURE SetCol(s: ScreenPtr;
num, R, G, B: INTEGER);

Diese Prozeduren setzen die Farben des Bildschirms auf den *s* zeigt. *SetColors* setzt dabei die Farben mit den Nummern *0* bis *LEN(cols)-1* auf die Werte von *cols*. Dabei entsprechen die Rot-, Grün- und Blauanteile der Ziffern einer dreistelligen Hexadezimalzahl. Gelb hat also den Wert *0FF0H*. Mit *SetCol* können die Rot-, Grün- und Blauanteile der Farbe *num* direkt verändert werden.

**PROCEDURE NumToRGB(num: INTEGER;
VAR r, g, b: INTEGER);
PROCEDURE RGBToNum(r, g, b: INTEGER): INTEGER;**

Mit diesen beiden Prozeduren kann ein Farbwert in die Rot-, Grün- und Blauanteile aufgespalten bzw. aus diesen Anteilen ein Farbwert berechnet werden.

**PROCEDURE FrontPen(d: DispElPtr; pen: SHORTINT);
PROCEDURE BackPen(d: DispElPtr; pen: SHORTINT);**

Die Zeichenfarbe (*FrontPen*) und die Hintergrundfarbe für Textausgabe (*BackPen*) kann mit diesen Funktionen auf die Farbe Nummer *pen* gesetzt werden.

**PROCEDURE Jam1(d: DispElPtr);
PROCEDURE Jam2(d: DispElPtr);**

Mit diesen Prozeduren kann der Zeichenmodus gewählt werden: Nach *Jam1* wird nur mit der Zeichenfarbe gezeichnet. Bei *Jam2* wird sowohl die Zeichen als auch die Hintergrundfarbe verwendet.

PROCEDURE Complement(d: DispElPtr);

Der Zeichenmodus wird auf *Complement* gesetzt. Beim Zeichnen wird dann die Nummer *x* der Farbe jedes Punktes durch *n-1-x* ersetzt, wobei *n* die Anzahl der Farben ist.

Mit diesem Modus kann das Zeichnen sehr leicht wieder rückgängig gemacht werden. Es muß lediglich nochmals mit *Complement* gezeichnet werden.

PROCEDURE LinePattern(d: DispElPtr; pat: INTEGER);

Beim Zeichnen von Linien kann man ein Muster verwenden.

Normalerweise ist dieses Muster so gesetzt, daß eine durchgezogene Linie gezeichnet wird. Als *pat* kann hier das gewünschte Muster gesetzt werden. Für häufige Muster sind in *Display* folgende Konstanten definiert:

```
line    : durchgezogene Linie
dots    : gepunktete Linie
bigdots : kurz gestrichelte Linie
broken  : gestrichelte Linie
```

PROCEDURE Clear(d: DispElPtr);

Die Zeichenfläche von *d* wird gelöscht.

PROCEDURE Line(d: DispElPtr; x1, y1, x2, y2: INTEGER);

Eine Linie von *(x1,y1)* nach *(x2,y2)* wird gezeichnet. Dabei wird das mit *LinePattern* gesetzte Muster beachtet.

PROCEDURE Dot(d: DispElPtr; x, y: INTEGER);

Der Punkt *(x,y)* wird gesetzt.

PROCEDURE DotColor(d: DispElPtr;
x, y: INTEGER): INTEGER;

Das Ergebnis ist die Nummer der Farbe des Punktes *(x,y)*.

PROCEDURE Rect(d: DispElPtr; x, y, w, h: INTEGER);
PROCEDURE Box(d: DispElPtr; x, y, w, h: INTEGER);

Das Rechteck mit der linken oberen Ecke an *(x,y)* und der Größe *w* und *h* wird gezeichnet. *Rect* zeichnet dabei nur die Kanten des Rechtecks, *Box* zeichnet ein ausgefülltes Rechteck. Bei *Rect* wird das mit *LinePattern* gesetzte Muster beachtet.

PROCEDURE Move(d: DispElPtr; x, y: INTEGER);
PROCEDURE Draw(d: DispElPtr; x, y: INTEGER);

Move setzt eine unsichtbare Zeichenmarke an die Position (*x,y*). *Draw* bewegt die Zeichenmarke an die Position (*x,y*) und zeichnet dabei eine Linie.

PROCEDURE Text(d: DispElPtr;
x, y: INTEGER;
s: ARRAY OF CHAR);

Der Text *s* wird an die Position (*x,y*) geschrieben.

PROCEDURE Circle(d: DispElPtr; x, y, r: INTEGER);
PROCEDURE Ellipse(d: DispElPtr; x, y, rx, ry: INTEGER);

Es wird ein Kreis (*Circle*) bzw. eine Ellipse (*Ellipse*) mit dem Mittelpunkt (*x,y*) gezeichnet. Der Radius wird durch *r* bzw. *rx* und *ry* festgelegt.

PROCEDURE Font(d: DispElPtr;
name: ARRAY OF CHAR;
height: INTEGER): BOOLEAN;

Der Zeichensatz für die Textausgabe wird auf den Zeichensatz mit dem Namen *name* (mit Endung '.font') und der Höhe *height* gesetzt. Das Ergebnis ist *TRUE* wenn dieser Zeichensatz geöffnet werden konnte.

PROCEDURE SetTurtlePos(d: DispElPtr; x, y: REAL);
PROCEDURE GetTurtlePos(d: DispElPtr; VAR x, y: REAL);

Diese Prozeduren für Turtle-Grafik setzen die Schildkröte an die Position (*x,y*) bzw. geben die aktuelle Position der Schildkröte zurück.

PROCEDURE SetTurtleDir(d: DispElPtr; dir: REAL);
PROCEDURE GetTurtleDir(d: DispElPtr): REAL;

Die Richtung, in die die Schildkröte wandert, wird auf *dir* gesetzt bzw. als Funktionsresultat zurückgegeben. Dabei bedeutet *0* ° nach oben, *90* ° nach links, *180* ° nach unten usw. Der Parametr *dir* wird im Gradmaß angegeben.

PROCEDURE SetPen(d: DispElPtr);
PROCEDURE LiftPen(d: DispElPtr);

Die Schildkröte setzt (*SetPen*) bzw. hebt (*LiftPen*) ihren Stift. Ist der Stift gesetzt, so zeichnet die Schildkröte eine Linie, wenn sie sich vorwärts bewegt.

PROCEDURE Forward(d: DispElPtr; s: REAL);

Die Schildkröte bewegt sich um die Strecke *s* vorwärts.

PROCEDURE TurnLeft(d: DispElPtr; alpha: REAL);
PROCEDURE TurnRight(d: DispElPtr; alpha: REAL);

Die Schildkröte dreht sich um den Winkel *alpha* nach links bzw. nach rechts.

PROCEDURE SetCursor(d: DispElPtr;
on: BOOLEAN): BOOLEAN;
PROCEDURE CursorOn(d: DispElPtr);
PROCEDURE CursorOff(d: DispElPtr);

Die Schreibmarke für Textausgabe wird angezeigt (*CursorOn*) bzw. nicht angezeigt (*CursorOff*).

SetCursor entspricht einem Aufruf von *CursorOn* oder *CursorOff*, je nach dem ob *on TRUE* oder *FALSE* ist. Das Ergebnis ist *TRUE* wenn die Schreibmarke vor dem Aufruf von *SetCursor* angezeigt wurde.

PROCEDURE Position(d: DispElPtr; x, y: INTEGER);

Die Position der Schreibmarke wird auf (*x,y*) gesetzt. Dabei wird die Position nicht in Punkten, sondern in Zeichen angegeben. Es muß dabei *0<=x<d.txtWidth* und *0<=y<d.txtHeight* gelten.

PROCEDURE Plain(d: DispElPtr);

Die Schriftattribute werden auf Normalschrift gesetzt.

PROCEDURE UnderLinedOn(d: DispElPtr);
PROCEDURE UnderLinedOff(d: DispElPtr);

Text unterstreichen wird ein- bzw. ausgeschaltet.

PROCEDURE BoldOn(d: DispElPtr);
PROCEDURE BoldOff(d: DispElPtr);

Fettschrift wird ein- bzw. ausgeschaltet.

PROCEDURE ItalicOn(d: DispElPtr);
PROCEDURE ItalicOff(d: DispElPtr);

Kursivschrift wird ein- bzw. ausgeschaltet.

PROCEDURE Home(d: DispElPtr);
PROCEDURE ClrHome(d: DispElPtr);

Die Schreibmarke wird in die linke obere Ecke gesetzt. *ClrHome* löscht dabei zusätzlich die Zeichenfläche.

PROCEDURE ScrollUp(d: DispElPtr);
PROCEDURE ScrollUpN(d: DispElPtr; n: INTEGER);

Der in *d* angezeigte Text wird um eine bzw. um *n* Zeilen nach oben geschoben.

PROCEDURE ScrollDown(d: DispElPtr);
PROCEDURE ScrollDownN(d: DispElPtr; n: INTEGER);

Der in *d* angezeigte Text wird um eine bzw. um *n* Zeilen nach unten geschoben.

PROCEDURE InsertLine(d: DispElPtr; n: INTEGER);

In der Textzeile *n* wird eine Leerzeile eingefügt. Die unter dieser Zeile angezeigten Zeilen werden dabei um eine Zeile nach unten geschoben.

PROCEDURE DeleteLine(d: DispElPtr; n: INTEGER);

Die Textzeile *n* wird entfernt und der in den Zeilen darunter angezeigte Text wird um eine Zeile nach oben geschoben. In der untersten Zeile entsteht eine Leerzeile.

PROCEDURE WriteLn(d: DispElPtr);

Die Schreibmarke wird an den Anfang der nächsten Zeile gesetzt.

PROCEDURE Write(d: DispElPtr; Str: ARRAY OF CHAR);

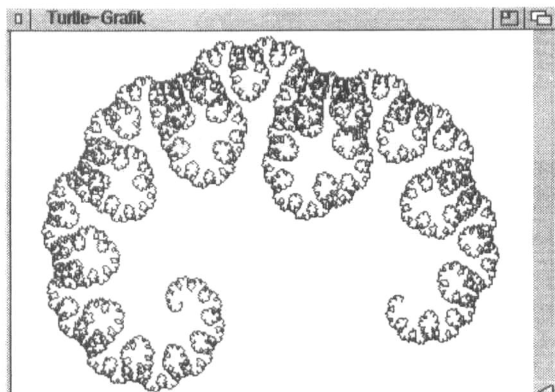
Der Text *Str* wird mit den eingestellten Attributen an der Position der Schreibmarke ausgegeben. Die Schreibmarke selbst wird danach hinter den ausgegebenen Texts gesetzt.

Beispiel

Mit den Prozeduren für Turtle-Grafik lassen sich fraktale Gebilde oft relativ leicht zeichnen. Ein Beispiel ist das Folgende Programm:

```
MODULE Turtle;
IMPORT D := Display, Dos;
VAR w: D.WindowPtr;
  PROCEDURE Draw(s: REAL);
  BEGIN
    IF s>2 THEN
      D.TurnLeft (w,36.9); Draw(s*0.8);
      D.TurnRight(w,90  ); Draw(s*0.6);
      D.TurnLeft (w,53.1);
    ELSE
      D.Forward(w,s);
    END;
  END Draw;
BEGIN
  NEW(w);
  IF D.OpenWindow(w,"Turtle-Grafik",
                  0,0,512,256,NIL) THEN
    D.SetTurtlePos(w,w.width/4,w.height/4*3);
    D.SetTurtleDir(w,-90); D.SetPen(w);
    D.Jam1(w); D.FrontPen(w,1); Draw(w.width/3);
  END;
  Dos.Delay(200);
END Turtle.
```

Die Prozedur *Draw* bewegt die Schildkröte um *s* vorwärts. Ist *s* klein, wird sie einfach geradeaus um *s* vorwärts bewegt. Sonst wird sie über einen 'Umweg', der zunächst etwas schräg nach links und dann wieder nach rechts führt, auch um diese Strecke vorwärts bewegt.



Die Ausgabe dieses Programms ist die im Bild rechts:

Eingaben in Display-Fenster

Das Modul *Display* bietet selbst keine Prozeduren zum Einlesen einer Eingabe über ein Fenster. Um über ein Display-Fenster Eingaben zu erhalten, müssen zunächst mit *Intuition.ModifyIDCMP* die gewünschten Eingabearten für das Intuition-Fenster gewählt werden. Der Zeiger auf die Intuition-Fensterstruktur ist in dem Recordelement *window* enthalten. Danach können mit *Exec.GetMsg* über den *userPort* des Intuition-Fensters Eingaben eingelesen werden. Da eine genaue Beschreibung den Rahmen dieses Handbuchs sprengen würde, sei hier auf [RKM: Libraries 92] und die Beispiele aus [AMOK] verwiesen.

FileReq

```
DEFINITION FileReq;

IMPORT I := Intuition;

VAR
  pattern : ARRAY 80 OF CHAR;
  defaultWidth, defaultHeight,
  defaultLeft, defaultTop     : INTEGER;

PROCEDURE FileReq(hail: ARRAY OF CHAR;
               VAR name: ARRAY OF CHAR): BOOLEAN;
PROCEDURE FileReqWin(hail: ARRAY OF CHAR;
               VAR name: ARRAY OF CHAR;
               win: I.WindowPtr): BOOLEAN;
PROCEDURE FileReqSave(hail: ARRAY OF CHAR;
               VAR name: ARRAY OF CHAR): BOOLEAN;
PROCEDURE FileReqWinSave(hail: ARRAY OF CHAR;
               VAR name: ARRAY OF CHAR;
               win: I.WindowPtr): BOOLEAN;

END FileReq.
```

Oft ist es nötig, einen Dateinamen vom Benutzer einzulesen. Dies geschieht am komfortabelsten über ein Dateiauswahlfenster. Diese bietet

das AmigaOS jedoch erst ab der Version 2.0 standardmäßig an. In früheren Versionen des Betriebssystems kann man sich jedoch mit dem Dateiauswahlfenster der frei kopierbaren 'arp.library' behelfen. Dieses Modul macht diese Unterscheidung automatisch. Die Prozeduren öffnen unter allen Versionen des AmigaOS ein Dateiauswahlfenster, jedoch bei Versionen vor 2.0 unter der Voraussetzung, daß sich die 'arp.library' im Verzeichnis 'LIBS:' befindet. Dir Prozeduren im Einzelnen:

PROCEDURE FileReq(hail: ARRAY OF CHAR;
VAR name: ARRAY OF CHAR): BOOLEAN;
PROCEDURE FileReqWin(hail: ARRAY OF CHAR;
VAR name: ARRAY OF CHAR;
win: I.WindowPtr): BOOLEAN;

Diese Prozeduren öffnen ein Dateiauswahlfenster mit dem Titel *hail. name* gibt dabei den Dateinamen an, der zunächst angezeigt werden soll und enthält nach dem Aufruf den eingegebenen Dateinamen. Das Ergebnis ist *TRUE* wenn das Fenster mit dem *Ok*-Symbol verlassen wurde, ansonsten *FALSE*. Der Parameter *win* bei *FileReqWin* gibt ein Fenster an, auf das sich das Dateiauswahlfenster beziehen soll. Das Dateiauswahlfenster wird auf dem gleichen Bildschirm wie *win* geöffnet. *win* kann *NIL* sein.

PROCEDURE FileReqSave(hail: ARRAY OF CHAR;
VAR name: ARRAY OF CHAR): BOOLEAN;
PROCEDURE FileReqWinSave(hail: ARRAY OF CHAR;
VAR name: ARRAY OF CHAR;
win: I.WindowPtr): BOOLEAN;

Diese Prozeduren entsprechen den oben beschriebenen Prozeduren *FileReq* bzw. *FileReqWin*. Sie öffnen jedoch ein Dateiauswahlfenster zum Speichern einer Datei. Dieses hat einen dunklen Hintergrund und bietet die Möglichkeit, neue Verzeichnisse zu anzulegen.

Über die globale Variable *pattern* kann ein Muster für die Dateinamen angegeben werden. Die Variablen *defaultWidth*, *defaultHeight*, *defaultLeft* und *defaultTop* geben die Maße und die Position des Dateiauswahlfensters an. Sie können vor dem Öffnen des ersten Fenster gesetzt werden, ansonsten sind sie mit Standardwerten vorbelegt. Nach dem Anzeigen eines Dateiauswahlfensters setzt *FileReq* diese Werte auf die evtl. veränderte Position und Größe des Fensters, so daß das nächste Dateiauswahlfenster an der gleichen Position geöffnet wird.

FileSystem

```
DEFINITION FileSystem;

IMPORT
  d := Dos,
  e := Exec,
  BT := BasicTypes,
  sys :=  SYSTEM;

CONST
  ok = 0;
  eof = 1;
  readerr = 2;
  writeerr = 3;
  onlyread = 4;
  onlywrite = 5;
  toofar = 6;
  outofmem = 7;
  cantopen = 8;
  cantlock = 9;

TYPE
  FilePtr = POINTER TO File;
  File = RECORD (BT.ANYDesc)
    handle : d.FileHandlePtr;
    status : INTEGER;
    write : BOOLEAN;
    read : BOOLEAN;
    name : ARRAY 256 OF CHAR;
```

```
    string : BT.DynString;
  END;

PROCEDURE Open(VAR file: File;
              name: ARRAY OF CHAR;
              write: BOOLEAN): BOOLEAN;
PROCEDURE OpenReadWrite(VAR file: File;
              name: ARRAY OF CHAR): BOOLEAN;
PROCEDURE Close(VAR file: File): BOOLEAN;
PROCEDURE Read(VAR file: File;
              VAR to: ARRAY OF sys.BYTE): BOOLEAN;
PROCEDURE ReadChar(VAR file: File;
              VAR ch: CHAR): BOOLEAN;
PROCEDURE ReadString(VAR file: File;
              VAR to: ARRAY OF CHAR): BOOLEAN;
PROCEDURE ReadLongString(VAR file: File): BOOLEAN;
PROCEDURE ReadBlock(VAR file: File;
              to: e.APTR;
              size: LONGINT): BOOLEAN;
PROCEDURE Write(VAR file: File;
              from: ARRAY OF sys.BYTE): BOOLEAN;
PROCEDURE WriteChar(VAR file: File;
              ch: CHAR): BOOLEAN;
PROCEDURE WriteString(VAR file: File;
              from: ARRAY OF CHAR): BOOLEAN;
PROCEDURE WriteBlock(VAR file: File;
              from: e.APTR;
              size: LONGINT): BOOLEAN;
PROCEDURE Size(VAR file: File): LONGINT;
PROCEDURE Position(VAR file: File): LONGINT;
PROCEDURE Move(VAR file: File;
              to: LONGINT): BOOLEAN;
PROCEDURE Forward(VAR file: File;
              to: LONGINT): BOOLEAN;
PROCEDURE Backward(VAR file: File;
              to: LONGINT): BOOLEAN;
PROCEDURE Delete(VAR file: File): BOOLEAN;
PROCEDURE Exists(name: ARRAY OF CHAR): BOOLEAN;

END FileSystem.
```

Dieses Modul erleichtert das Arbeiten mit Dateien. Um mit diesem Modul zu arbeiten, muß man zunächst eine Variable des Typs *File* deklarieren. Mit dieser Variablen kann dann mit *Open* eine Datei geöffnet werden, die dann mit den restlichen Prozeduren bearbeitet und mit *Close* wieder geschlossen werden kann.

Alle Prozeduren mit einem Ergebnis des Typs *BOOLEAN* zeigen ihre erfolgreiche Ausführung durch das Resultat *TRUE* an. In diesem Fall wird *File.status* auf *ok* gesetzt. Trat ein Fehler auf, so ist das Ergebnis *FALSE* und in *File.status* wird eine der Fehlernummern *eof* bis *cantlock* eingetragen. Der Wert von *status* erlaubt eine genauere Untersuchung der Fehlerursache. Die Werte haben dabei die folgenden Bedeutungen:

```
ok          kein Fehler
eof         das Dateiende ist erreicht
readerr     Lesefehler
writeerr    Schreibfehler
onlyread    Datei darf nur gelesen werden
onlywrite   In Datei darf nur geschrieben werden
toofar      Sprung über Dateiende bzw. Anfang
outofmem    Speichermangel
cantopen    Konnte Datei nicht öffnen
cantlock    Lock auf Datei war nicht möglich
```

Die Prozeduren von *FileSystem* sind:

PROCEDURE Open(VAR file: File;
name: ARRAY OF CHAR;
write: BOOLEAN): BOOLEAN;

Die Datei *name* wird geöffnet. Ist *write TRUE* so wird eine neue Datei erzeugt und zum Schreiben geöffnet. Auf diese Datei sind Lesende Zugriffe nicht möglich. Andernfalls wird eine Datei zum Lesen geöffnet, in die nicht geschrieben werden kann.

**PROCEDURE OpenReadWrite(VAR file: File;
name: ARRAY OF CHAR): BOOLEAN;**

Die Datei *name* wird zum gemischten lesenden und schreibendem Zugriff geöffnet.

PROCEDURE Close(VAR file: File): BOOLEAN;

Die Datei *file*, die zuvor erfolgreich geöffnet wurde, wird geschlossen. Bei Dateien, die auch zum Schreiben geöffnet wurden, kann *Close* fehlschlagen, da beim Schließen eventuell noch Daten in die Datei geschrieben werden müssen. Bei Dateien, die lediglich zum Lesen geöffnet wurden, kann das Funktionsresultat von *Close* ignoriert werden.

**PROCEDURE Read(VAR file: File;
VAR to: ARRAY OF sys.BYTE): BOOLEAN;**

Aus der Datei *file* werden *LEN(to)* Bytes gelesen und in *to* gespeichert. Als *to* können beliebige Variablen übergeben werden.

Vorsicht ist hier jedoch angebracht, wenn die übergebene Variable verfolgte Zeiger enthält. Dann muß sichergestellt sein, daß die verfolgten Zeiger nur mit *NIL* beschrieben werden, da alle anderen Werte sicher nicht auf ein Objekt des Garbage-Collectors zeigen und somit sicher zu einem Programmabsturz führen.

**PROCEDURE ReadChar(VAR file: File;
VAR ch: CHAR): BOOLEAN;**

Das nächste Zeichen wird aus der Datei gelesen und in *ch* geschrieben.

PROCEDURE ReadString(VAR file: File;
VAR to: ARRAY OF CHAR): BOOLEAN;

Eine Zeichenkette die mit 0X (nul) oder 0AX (line feed) beendet wird, wird aus *file* gelesen und in *to* gespeichert. Dabei werden jedoch höchstens *LEN(to)* Zeichen gelesen.

PROCEDURE ReadLongString(VAR file: File): BOOLEAN;

Eine beliebig lange Zeichenkette, die mit 0X oder 0AX endet, wird eingelesen. Sie wird in *file.string^* gespeichert. Wird diese Prozedur mehrmals aufgerufen, so wird beim ersten Aufruf Speicher für die Zeichenkette angefordert, der bei den folgenden Aufrufen weiter verwendet wird. Lediglich beim Einlesen längerer Zeichenketten als die vorigen wird neu Speicher angefordert.

PROCEDURE ReadBlock(VAR file: File;
to: e.APTR;
size: LONGINT): BOOLEAN;

Dies ist eine 'niedrige' Prozedur zum Einlesen von Daten von der Datei. In Oberon-Programmen sollte besser *Read* verwendet werden, *ReadBlock* wird jedoch manchmal bei der systemnahen Programmierung nötig. Es werden *size* Bytes aus der Datei *file* an die Speicheradresse *to* eingelesen.

PROCEDURE Write(VAR file: File;
from: ARRAY OF sys.BYTE): BOOLEAN;

Die als *from* übergebene Variable wird in der Datei gespeichert. Da *from* jeden beliebigen Typ haben darf, kann hier jede Variable übergeben werden.

**PROCEDURE WriteChar(VAR file: File;
ch: CHAR): BOOLEAN;**

Das Zeichen *ch* wird in die Datei geschrieben.

**PROCEDURE WriteString(VAR file: File;
from: ARRAY OF CHAR): BOOLEAN;**

Die Zeichenkette *from* wird in der Datei gespeichert. Nach der Zeichenkette wird als Ende-Kennung ein 0AX (line feed) in die Datei geschrieben.

**PROCEDURE WriteBlock(VAR file: File;
from: e.APTR;
size: LONGINT): BOOLEAN;**

Dies ist eine 'niedrige' Prozedur zum Schreiben von Daten in die Datei. In Oberon-Programmen sollte besser *Write* verwendet werden, *WriteBlock* wird jedoch manchmal bei der systemnahen Programmierung nötig. Es werden *size* Bytes ab der Speicheradresse *to* in die Datei *file* geschrieben.

PROCEDURE Size(VAR file: File): LONGINT;

Das Ergebnis ist die Größe der Datei in Bytes.

PROCEDURE Position(VAR file: File): LONGINT;

Die aktuelle Byte-Position in der Datei wird zurückgegeben.

PROCEDURE Move(VAR file: File; to: LONGINT): BOOLEAN;

Die aktuelle Position in der Datei wird auf *to* gesetzt. *to* muß dazu zwischen Null und der Dateigröße liegen.

**PROCEDURE Forward(VAR file: File;
to: LONGINT): BOOLEAN;
PROCEDURE Backward(VAR file: File;
to: LONGINT): BOOLEAN;**

Die aktuelle Position wird um *to* weiter (*Forward*) bzw. zurück (*Backward*) bewegt.

PROCEDURE Delete(VAR file: File): BOOLEAN;

Die Datei *file* wird geschlossen und gelöscht. Diese Prozedur kann benutzt werden, um eine Datei nach einem Fehler zu entfernen.

PROCEDURE Exists(name: ARRAY OF CHAR): BOOLEAN;

Das Ergebnis ist *TRUE* wenn eine Datei mit den Namen *name* existiert. Anders als bei den anderen Prozeduren zeigt hier das Ergebnis *FALSE* keinen Fehler an, sondern nur das Fehlen der angegebenen Datei.

io

```
DEFINITION io;

IMPORT
  d := Dos,
  e := Exec;

VAR
  out, in : d.FileHandlePtr;
  Me : d.ProcessPtr;
  closeDelay : LONGINT;

PROCEDURE Write(ch: CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteString(str: ARRAY OF CHAR);
```

```
PROCEDURE Tab(n: INTEGER);
PROCEDURE Clear();
PROCEDURE WriteInt(x: LONGINT; n: INTEGER);
PROCEDURE WriteHex(x: LONGINT; n: INTEGER);
PROCEDURE Read(VAR ch: CHAR);
PROCEDURE ReadString(VAR str: ARRAY OF CHAR);
PROCEDURE ReadInt(VAR x: LONGINT): BOOLEAN;
PROCEDURE ReadInteger(VAR x: INTEGER): BOOLEAN;
PROCEDURE ReadShortInt(VAR x: SHORTINT): BOOLEAN;
PROCEDURE ReadHex(VAR x: LONGINT): BOOLEAN;
PROCEDURE ReadIntOk(VAR x: LONGINT);
PROCEDURE ReadIntegerOk(VAR x: INTEGER);
PROCEDURE ReadShortIntOk(VAR x: SHORTINT);
PROCEDURE ReadHexOk(VAR x: LONGINT);
PROCEDURE Format(str: ARRAY OF CHAR;
                 data: e.APTR);

END io.
```

io stellt grundlegende Ein- und Ausgaberoutinen für Text zur Verfügung. Nach dem Start von der Shell benutzt *io* das Shell-Fenster für die Ein- und Ausgaben (wenn diese nicht umgeleitet wurdne). Beim Start von der Workbench öffnet *io* ein Console-Fenster ([RKM: Devices 91]) für diese Aufgabe. Über das Merkmal *WINDOW* des Piktogramms des Programms kann dabei das Ausssehen dieses Fensters beeinflußt werden.

Die Variablen *in* und *out* beinhalten die *FileHandles* der Ein- und Ausgabefenster bzw. -dateien des Programms, genaueres hierzu ist in [AmigaDos 91] zu finden. *closeDelay* gibt die Zeit an, die das bei einem Workbench-Start geöffnete Console-Fenster nach dem Ablauf des Programms geöffnet bleiben soll. Die Zeit wird in fünzigstel Sekunden angegeben. Gewöhnlich ist dieser Wert auf *50*, so daß der Benutzer nach dem Programmablauf noch kurz für eine Sekunde die Ausgabe des Programms zu sehen bekommt. Soll das Fenster sofort geschlossen werden, muß *closeDelay* auf Null gesetzt werden.

Der Zeiger *Me* zeigt auf die Prozeßstruktur des Programms.

Die Prozeduren von io werden im folgenden genau beschrieben:

PROCEDURE Write(ch: CHAR);

Das Zeichen *ch* wird im Ausgabefenster ausgegeben.

PROCEDURE WriteLn;

Das Zeichen *ASCII.lf* (line feed) wird ausgegeben. Dadurch wird die Schreibmarke an den Anfang der nächsten Zeile gesetzt.

PROCEDURE WriteString(str: ARRAY OF CHAR);

Die Zeichenkette *str* wird ausgegeben.

PROCEDURE Tab(n: INTEGER);

Die Schreibmarke wird um *n* (Leer-) Zeichen weiter nach rechts gesetzt.

PROCEDURE Clear();

Der Inhalt des Ausgabefensters wird gelöscht und die Schreibmarke wird in die linke obere Ecke des Fensters gesetzt.

PROCEDURE WriteInt(x: LONGINT; n: INTEGER);
PROCEDURE WriteHex(x: LONGINT; n: INTEGER);

Die Zahl *x* wird als *n*-stellige Dezimal- (*WriteInt*) bzw. Hexadezimalzahl (*WriteHex*) ausgegeben. Die Dezimalzahl wird dabei evtl. mit Vorzeichen rechtsbündig ausgegeben, die Hexadezimalzahl wird mit führenden Nullen angezeigt.

PROCEDURE Read(VAR ch: CHAR);

Ein Zeichen wird eingelesen und in *ch* gespeichert.

PROCEDURE ReadString(VAR str: ARRAY OF CHAR);

Eine mit der Returntaste beendete Zeichenkette wird eingelesen und in *str* gespeichert.

PROCEDURE ReadInt(VAR x: LONGINT): BOOLEAN;
PROCEDURE ReadInteger(VAR x: INTEGER): BOOLEAN;
PROCEDURE ReadShortInt(VAR x: SHORTINT): BOOLEAN;

Es wird eine *LONGINT-* (*ReadInt*), *INTEGER* (*ReadInteger*) bzw. *SHORTINT*-Zahl (*ReadShortInt*) eingelesen und in *x* gespeichert. Wurde keine korrekte oder eine zu große Dezimalzahl eingegeben, so ist das Ergebnis *FALSE*.

PROCEDURE ReadHex(VAR x: LONGINT): BOOLEAN;

Es wird eine Hexadezimalzahl eingelesen und in *x* gespeichert. Dabei werden Groß- und Kleinbuchstaben akzeptiert. Das Ergebnis ist *TRUE* wenn eine korrekte Zahl eingegeben wurden.

PROCEDURE ReadIntOk(VAR x: LONGINT);
PROCEDURE ReadIntegerOk(VAR x: INTEGER);
PROCEDURE ReadShortIntOk(VAR x: SHORTINT);
PROCEDURE ReadHexOk(VAR x: LONGINT);

Diese drei Prozeduren entsprechen den Prozeduren *ReadInt*, *ReadInteger*, *ReadShortInt* und *ReadHexOk*. Sie haben jedoch kein Funktionsergebnis. Eine falsche Eingabe führt zu einem undefinierten *x*.

PROCEDURE Format(str: ARRAY OF CHAR; data: e.APTR);

Dies Prozedur formatiert eine Zeichenkette mit der Prozedur *RawDoFmt* aus Exec. *data* zeigt dabei auf die zusätzlichen Daten, die in der Zeichenkette angezeigt werden sollen. Die formatierte Zeichenkette wird dann wie bei *WriteString*

ausgegeben. Genaueres zu den Formatierungsmöglichkeiten ist in [RKM: Autodocs 91] nachzulesen, hier sei nur ein Beispiel gegeben: Das Programm

```
MODULE Test;
IMPORT io, SYSTEM;
VAR
  format: STRUCT
    string: y.ADDRESS;
    hex: LONGINT;
  END;
BEGIN
  format.string := y.ADR("Zeichenkette");
  format.hex    := 1000000;
  io.Format("String: %s  hex = %08lx\n",
             SYSTEM.ADR(format));
END Test.
```

erzeugt die Ausgabe:

```
String: Zeichenkette  hex = 000F4240
```

Es ist darauf zu achten, daß die Ausgabe niemals länger als 255 Zeichen wird.

LongRealInOut

```
DEFINITION LongRealInOut;

PROCEDURE WriteReal(r: LONGREAL;
                    v, n: INTEGER;
                    exp: BOOLEAN): BOOLEAN;
PROCEDURE ReadReal(VAR r: LONGREAL): BOOLEAN;

END LongRealInOut.
```

Die Prozeduren dieses Moduls ermöglichen die Ausgabe von *LONGREAL*-Zahlen. Das Modul benutzt zur Umwandlung der Zahlen

in Zeichenketten die Routinen des Moduls *LongRealConversions* (Kapitel 20). Die Prozeduren sind:

PROCEDURE WriteReal(r: LONGREAL;
v, n: INTEGER;
exp: BOOLEAN): BOOLEAN;

Die *LONGREAL*-Zahl *r* wird in das Ausgabefenster von *io* (siehe oben) ausgegeben. Dabei werden *v* Vorkomma- und *n* Nachkommatstellen angezeigt. Ist exp *TRUE* so wird auch ein Exponent ausgegeben. Das Ergebnis ist *TRUE* wenn *r* in dem gewünschten Format angezeigt werden konnte.

PROCEDURE ReadReal(VAR r: LONGREAL): BOOLEAN;

Es wird eine *LONGREAL*-Zahl eingelesen. Die eingegebene Zahl darf dabei aus einem Vorzeichen, den Vorkommastellen, einem Punkt gefolgt von den Nachkommastellen und einem Exponenten bestehen. Der Exponent besteht aus einem *"E"* gefolgt von einem optionalen Vorzeichen und dem Wert des Exponenten. Das Ergebnis ist *TRUE* wenn eine korrekte Zahl eingegeben wurde.

RealInOut

```
DEFINITION RealInOut;

PROCEDURE WriteReal(r: REAL;
                    v, n: INTEGER;
                    exp: BOOLEAN): BOOLEAN;
PROCEDURE ReadReal(VAR r: REAL): BOOLEAN;

END RealInOut.
```

Die Prozeduren aus *RealInOut* entsprechen exakt denen aus *LongRealInOut* (siehe oben). Sie werden daher hier nicht erneut beschrieben.

Requests

```
DEFINITION Requests;

IMPORT I := Intuition;

PROCEDURE Request(head, msg,
               pos, neg: ARRAY OF CHAR): BOOLEAN;
PROCEDURE RequestWin(head, msg,
               pos, neg: ARRAY OF CHAR;
               win: I.WindowPtr): BOOLEAN;
PROCEDURE Assert(cc: BOOLEAN; msg: ARRAY OF CHAR);
PROCEDURE Fail(msg: ARRAY OF CHAR);
PROCEDURE BreakPoint(msg: ARRAY OF CHAR);

END Requests.
```

Dieses Modul ermöglicht es, einfache Dialogfenster leicht zu erzeugen. Dazu wird die Prozedur *AutoRequest* aus Intuition ([RKM: Libraries 92]) verwendet. *Requests* setzte die Prozedurvariable *OutOfMemHandler* des Moduls *OberonLib* auf eine Prozedur, die den Benutzer in einem Dialogfenster über den Speichermangel informiert und ihm die Wahl zwischen der Wiederholung der Speicheranforderung und einem Programmabbruch läßt. *Requests* exportiert die folgenden Prozeduren:

PROCEDURE Request(head, msg,
pos, neg: ARRAY OF CHAR): BOOLEAN;

Es wird ein Dialogfenster geöffnet. Das Fenster besteht aus zwei Textzeilen *head* und *msg* die den Grund des Erscheinens dieses Fensters kurz und einleuchtend beschreiben sollten. Zudem besteht das Dialogfenster aus einem oder zwei anwählbaren Symbolen *pos* und *neg* die das Dialogfenster mit einer positiven bzw. negativen Antwort verlassen. Dialogfenster, die lediglich eine Nachricht für den Benutzer beinhalten, jedoch keine Entscheidung von ihm verlangen, benötigen nur eine Symbol. Um das

zweite zu unterdrücken muß für *neg* eine leere Zeichenkette "" angegeben werden.

Beispiel:

```
IF Requests.Request("OEd Request:",
                    "Text verändert! Speichern?",
                    "Oh ja!","Nein Danke!")
THEN
  Save(text)
END;
```

PROCEDURE RequestWin(head, msg,
pos, neg: ARRAY OF CHAR;
win: I.WindowPtr): BOOLEAN;

Diese Prozedur entspricht *Request*. Hier kann jedoch zusätzlich ein Fenster *win* angegeben werden, auf das sich das Dialogfenster beziehen soll. Das Dialogfenster wird dann auf dem Bildschirm, auf dem sich *win* befindet, geöffnet. *win* kann *NIL* sein.

PROCEDURE Assert(cc: BOOLEAN; msg: ARRAY OF CHAR);

Mit *Assert* kann ein Programm leicht bei einem Fehler abgebrochen werden. Ist *cc* erfüllt, also *TRUE* so kehrt *Assert* ohne etwas zu tun zurück. Ansonsten wird *msg* in einem Dialogfenster angezeigt. Wurde das Programm von einer Shell aus gestartet so wird kein Dialogfenster geöffnet. Stattdessen wird *msg* in dem Shell-Fenster ausgegeben. Danach wird das Programm mit *HALT(20)* beendet.

Beispiel:

```
window := Intuition.OpenWindow(newwindow);
Requests.Assert(window#NIL,
                "Konnte Fenster nicht öffnen");
```

PROCEDURE Fail(msg: ARRAY OF CHAR);

Fail(msg) ist eine Abkürzung für *Assert(FALSE,msg)*.

PROCEDURE BreakPoint(msg: ARRAY OF CHAR);

BreakPoint kann während der Testphase von Programmen verwendet werden, um Fehler zu lokalisieren. Es öffnet ein Dialogfenster mit der Meldung *msg*. Dabei kann man wählen, ob im Programm fortgefahren oder ob das Programm abgebrochen werden soll.

23. Modulbibliothek: Standardmodule



Damit die in [Reiser 92] beschriebenen Oberon-Programme mit Amiga Oberon übersetzt werden können, bietet die Modulbibliothek die dort vorgestellten Standardmodule. Dies sind zwei einfache Module für die Ein- und Ausgabe und ein Modul für Grafikausgabe.

In

```
DEFINITION In;

VAR
  Done : BOOLEAN;

PROCEDURE Open;
PROCEDURE Char(VAR ch: CHAR);
PROCEDURE Int(VAR i: INTEGER);
PROCEDURE LongInt(VAR i: LONGINT);
PROCEDURE Real(VAR x: REAL);
PROCEDURE Name(VAR name: ARRAY  OF CHAR);
PROCEDURE String(VAR str: ARRAY  OF CHAR);

END In.
```

Dieses Modul stellt einfache Eingabeprozeduren ähnlich denen von *io* zur Verfügung. Die Variable *Done* wird auf *FALSE* gesetzt, sobald eine Eingabeprozedur fehlschlägt. Die danach aufgerufenen Eingabeprozeduren lesen dann nichts mehr ein und belassen *Done* auf *FALSE*. Nur durch einen Aufruf von *Open* wird *Done* wieder auf *TRUE* gesetzt. Die Prozeduren von *In* sind die im folgenden beschriebenen:

PROCEDURE Open;

In dieser Implementierung macht *Open* nichts anderes als *Done* auf *TRUE* zu setzen.

PROCEDURE Char(VAR ch: CHAR);

Das nächste Zeichen wird eingelesen und in *ch* gespeichert.

PROCEDURE Int(VAR i: INTEGER);
PROCEDURE LongInt(VAR i: LONGINT);
PROCEDURE Real(VAR x: REAL);

Diese Prozeduren lesen *INTEGER*-, *LONGINT*- bzw. *REAL*-Zahlen ein und speichern sie in ihrem Referenzparameter.

PROCEDURE Name(VAR name: ARRAY OF CHAR);

Es wird ein Name eingelesen und in *name* gespeichert. Ein Name ist eine Folge von Bezeichnern und Punkten.

PROCEDURE String(VAR str: ARRAY OF CHAR);

Alle Zeichen nach dem nächsten Leerzeichen oder Zeilenumbruch werden bis zum nächsten Leerzeichen bzw. Zeilenumbruch eingelesen.

Out

```
DEFINITION Out;

PROCEDURE Open;
PROCEDURE Char(ch: CHAR);
PROCEDURE Ln;
PROCEDURE Int(i, n: LONGINT);
PROCEDURE Real(x: REAL; n: INTEGER);
PROCEDURE String(str: ARRAY  OF CHAR);

END Out.
```

Als Gegenstück zu In bietet dieses Modul einfache Prozeduren zur Ausgabe. Diese sind:

PROCEDURE Open;

Der Inhalt des Ausgabefensters wird gelöscht und die Schreibmarke wird in die linke obere Ecke des Fensters gesetzt.

PROCEDURE Char(ch: CHAR);

Das Zeichen ch wird ausgegeben.

PROCEDURE Ln;

Die Schreibmarke wird an den Anfang der nächsten Zeile gesetzt.

PROCEDURE Int(i, n: LONGINT);
PROCEDURE Real(x: REAL; n: INTEGER);

Die Zahl i bzw. x wird mit als n-stellige Dezimalzahl bzw. als Dezimalbruch mit n Ziffern ausgegeben.

PROCEDURE String(str: ARRAY OF CHAR);

Die Zeichenkette str wird ausgegeben.

XYplane

```
DEFINITION XYplane;

CONST
   erase = 0;
   draw = 1;

VAR
   X, Y, W, H : INTEGER;

PROCEDURE Clear;
PROCEDURE Open;
```

```
PROCEDURE Dot(x, y, mode: INTEGER);
PROCEDURE IsDot(x, y: INTEGER): BOOLEAN;
PROCEDURE Key(): CHAR;

END XYplane.
```

Dieses Modul stellt eine sehr einfache Abstraktion einer Zeichenfläche dar. Die rechteckige Fläche besteht aus einzelnen rechteckigen Punkten die entweder gesetzt oder gelöscht sein können. Die Punkte werden in einem kartesischen Koordinatensystem mit dem Ursprung in der linken unteren Ecke und positiven Koordinatenachsen nach oben und rechts angeordnet.

Mit der Prozedur *Open* wird eine solche Fläche in Form eines Fensters angelegt. Die Variablen *X* und *Y* enthalten dann die Koordinaten der linken unteren Ecke und über *W* und *H* kann die Breite und Höhe der Zeichenfläche ausgelesen werden.

Mit den folgenden Prozeduren kann der Inhalt der Zeichenfläche verändert werden:

PROCEDURE Clear;

Alle Punkte der Zeichenfläche werden gelöscht.

PROCEDURE Open;

Ein neues Fenster mit einer gelöschten Zeichenfläche wird geöffnet.

PROCEDURE Dot(x, y, mode: INTEGER);

Je nach dem, ob für den Parameter *mode* der Wert *draw* oder *erase* angegeben wurde, wird der Punkt mit den Koordinaten *x* und *y* gesetzt bzw. gelöscht.

PROCEDURE IsDot(x, y: INTEGER): BOOLEAN;

Das Ergebnis ist *TRUE* wenn der Punkt mit den Koordinaten *x* und *y* gesetzt ist, ansonsten ist es *FALSE*.

PROCEDURE Key(): CHAR;

Diese Prozedur liest das Zeichen von der Tastatur, das zuletzt vor dem Aufruf von *Key* eingegeben wurde. Wurde kein Zeichen eingegeben so ist das Ergebnis 0X.

Wurde das Schließsymbol des Fensters, das die Grafikfläche enthält, angewählt, so ist das Ergebnis von *Key* "Q", so daß ein Programm, das *XYplane* verwendet, hierauf sinnvoll reagieren kann.

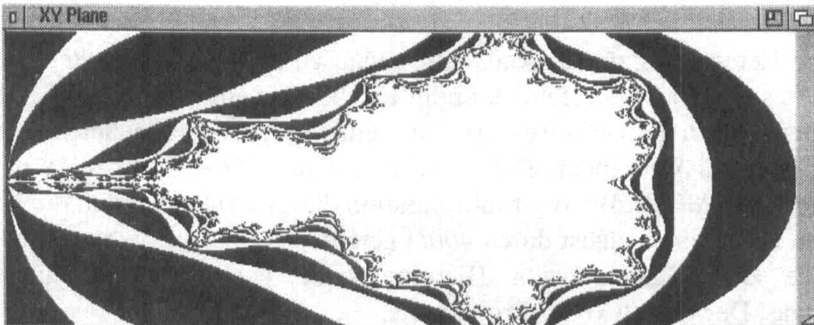
Beispielprogramm: Das folgende Programm zeichnet eine fraktale Grafik, ein sogenanntes 'Apfelmännchen' mit Hilfe des Moduls *XYplane*. Dabei kann das Zeichnen jederzeit durch Drücken von 'Q' oder mit dem Schließsymbol abgebrochen werden.

Die Berechnung des Fraktals hier genau zu erklären würde den Sinn dieses Handbuchs verfehlen. Lediglich die verwendeten Zahlen seinen kurz erklärt: Für eine bessere Effizienz werden statt reellen Zahlen *LONGINTs* verwendet. Dabei entspricht die *LONGINT*-Zahl *x* dem Wert *x/100000H*. Bei der Multiplikation dieser Zahlen werden die beiden Faktoren zunächst durch *400H* geteilt, so daß als Ergebnis wieder eine solche Zahl entsteht. Hierunter leidet jedoch die Genauigkeit etwas. Der Quelltext des Programms:

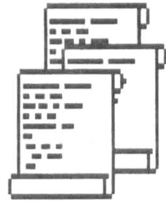
```
MODULE Mandel;
IMPORT xy : XYplane;
VAR
  zr,zi,ar,ai,dr,di,sr,si,st: LONGINT;
  x, y, i: INTEGER;
BEGIN
```

```
  sr := 300000H DIV xy.W; si := 200000H DIV xy.H;
  st := -2*100000H; zi := 100000H;
  FOR y := 0 TO xy.H-1 DO
    IF CAP(xy.Key())="Q" THEN HALT(0) END;
    DEC(zi,si); zr := st;
    FOR x := 0 TO xy.W-1 DO
      i := 0; ar := zr; ai := zi;
      REPEAT
        dr := ar DIV 400H; di := ai DIV 400H;
        ai := 2 * dr * di + zi;
        dr := dr*dr; di := di*di;
        ar := dr - di + zr;
        INC(i)
      UNTIL (i > 17) OR (dr + di > 400000H);
      xy.Dot(x,y,i MOD 2); INC(zr,sr);
    END;
  END;
END Mandel.
```

Und dies ist die erzeugte Grafik:



24. Modulbibliothek: Multitasking



Eine der herausragenden Fähigkeiten des Amiga ist das Multitasking. Dabei können nicht nur mehrere Programme gleichzeitig abgearbeitet werden, sondern auch einzelne Programme können für verschiedene Aufgaben mehrere Prozesse starten. Damit dies auch in Oberon-Programmen leicht und sicher (Garbage-Collector) möglich ist, enthält die Modulbibliothek das Modul *Concurrency*.

Concurrency

```
DEFINITION Concurrency;

IMPORT BT := BasicTypes, Dos;

TYPE
  ProcessProc = PROCEDURE(data: BT.ANY): BT.ANY;
  Process = POINTER TO ProcessDesc;
  ProcessDesc = RECORD (BT.ANYDesc)
    dosProcess : Dos.ProcessPtr;
    PROCEDURE (p:Process) Wait(): BT.ANY;
    PROCEDURE (p:Process) isRunning(): BOOLEAN;
  END;

PROCEDURE NewProcess(proc: ProcessProc;
                     data: BT.ANY): Process;
PROCEDURE NewProcessX(proc: ProcessProc;
                     data: BT.ANY;
                     stackSize: LONGINT;
                     priority: SHORTINT): Process;
PROCEDURE WaitForAllProcesses;

END Concurrency.
```

Globale Prozeduren, die zuweisungskompatibel zum Typ *ProcessProc* sind, können als eigenständige Prozesse gestartet werden. Diese Pro-

zeduren müssen reentrant sein, das heißt, sie dürfen auf keine globalen Daten oder Variablen zugreifen, während dies auch ein anderer Prozeß könnte. Dies wird am einfachsten dadurch sichergestellt, daß eine solche Prozedur und alle Prozeduren, die durch sie aufgerufen werden, überhaupt nicht auf globale Variablen und Daten zugreifen. Da dies jedoch manchmal mit unnötig viel Aufwand verbunden ist, können die globalen Daten mit Semaphoren geschützt werden. Diese stellt das AmigaOS zur Verfügung, eine Beschreibung befindet sich in [RKM: Libraries 92].

Die meisten Standardmodule, die Prozeduren zur Ein- oder Ausgabe anbieten, sind nicht reentrant und dürfen daher von neuen Prozessen nicht verwendet werden! Eine Ausnahme bildet das Modul *Display*, hier ist es erlaubt, daß die Prozeduren von mehreren Prozessen gleichzeitig benutzt werden, solange keine zwei Prozesse gleichzeitig dasselbe Fenster oder denselben Bildschirm benutzen.

Dieses Module enthält keine Prozeduren für die Inter-Prozeß-Kommunikation. Sehr leistungsfähige Methoden hierzu, wie Signale und Nachrichten, bietet die Exec-Library des AmigaOS. Auch hier sei auf [RKM: Libraries 92] verwiesen.

Für jeden mit *Concurrency* gestarteten Prozeß wird ein Objekt des Typs *Process* erzeugt. Dieses Record enthält einen Zeiger *dosProcess* auf die Prozeß-Struktur, wie sie von der Dos- und der Exec-Library verwendet wird. Dieser Zeiger wird für die Kommunikation zwischen den Prozessen benötigt.

Concurrency stellt die im folgenden beschriebenen Prozeduren zur Verfügung:

PROCEDURE NewProcess(proc: ProcessProc;
data: BT.ANY): Process;

Die Prozedur *proc* wird als neuer Prozeß gestartet. Der Parameter *data* kann dazu benutzt werden, Parameter an *proc* zu

übergeben. Dazu muß eine Erweiterung von *BasicTypes.ANY-Desc* definiert werden, die die gewünschten Daten enthält. Mit Hilfe eines Typeguards kann in proc dann auf diese Daten zugegriffen werden. Wird kein Parameter benötigt, kann als *data* einfach *NIL* übergeben und innerhalb von *proc* der Parameter ignoriert werden.

Das Ergebnis von *NewProcess* ist ein Zeiger auf das Prozeßobjekt des neuen Prozesses. Konnte der Prozeß nicht gestartet werden, weil z.B. zu wenig Speicher zu Verfügung stand, ist das Ergebnis *NIL*. Über die typgebundenen Prozeduren des Prozeßobjekts kann der Zustand des Prozesses abgefragt und auf das Ende des Prozesses gewartet werden.

Damit *proc* auch selbst Objekte erzeugen und mit Zeigern arbeiten kann, teilt *NewProcess* dem Garbage-Collector mit, daß *proc* als neuer Mutator gestartet wurde (siehe Kapitel 16).

Der neue Prozeß erbt die Priorität von dem Prozeß, aus dem *NewProcess* aufgerufen wurde. Für den Stapelspeicher wird die Größe des Stapelspeichers des Hauptprogramms verwendet.

NewProcess ist reentrant, es kann also auch von einem mit *NewProcess* gestartetem Prozeß aus aufgerufen werden, um weitere Prozesse zu starten.

Beendet wird der Prozeß, wenn *proc* mit *RETURN* beendet wird. Dabei darf nicht vergessen werden, daß *proc* eine Funktion ist, die ein Ergebnis liefern muß. Dieses Ergebnis wird von *Wait* (siehe unten) zurückgegeben. Wird es nicht benötigt, kann es einfach *NIL* sein.

Ein Prozeß kann auch mit der Standardprozedur *HALT* beendet werden. Dann ist das Ergebnis des Prozesses auf jeden Fall *NIL* (siehe unten: *Wait*). Ein *HALT* beendet nur den betroffenen

Prozeß, keine anderen Prozesse. Insbesondere läuft der Hauptprozeß des Programms weiter.

Ein Laufzeitfehler in einem mit *NewProcess* gestarteten Prozeß führt auch zu einem Stop des betroffenen Prozesses. Der Fehler wird als Alert-Meldung angezeigt, wenn das Modul *NoGuru* importiert wurde (Kapitel 25). Das Ergebnis des Prozesses ist auch dann *NIL*. Alle anderen Prozesse bleiben von dem Laufzeitfehler unbeeinflußt.

Es ist wichtig, daß ein Prozeß, der auf globale Daten eines Moduls zugreift, beendet wird, bevor der CLOSE-Anweisungsteil des Moduls ausgeführt wird. In den CLOSE-Anweisungen von *Concurrency* wird darauf gewartet, daß alle Prozesse beendet werden. Ein endlos laufender Prozeß führt also dazu, daß das Programm niemals beendet werden kann.

PROCEDURE NewProcessX(proc: ProcessProc;
data: BT.ANY;
stackSize: LONGINT;
priority: SHORTINT): Process;

Wie *NewProcess* startet auch *NewProcessX* einen neuen Prozeß. Hier kann jedoch über die Parameter *stackSize* und *priority* die Größe des Stapelspeichers und die Priorität des neuen Prozesses angegeben werden.

PROCEDURE WaitForAllProcesses;

Diese Prozedur wartet solange, bis alle mit *NewProcess* gestarteten Prozesse beendet sind. Diese Prozedur kann zu Beginn einer CLOSE-Anweisung verwendet werden, um sicherzustellen, daß alle Prozesse beendet sind, bevor die CLOSE-Anweisung ausgeführt wird.

PROCEDURE (p: Process) Wait(): BT.ANY;

Dies Prozedur kehrt erst dann wieder zurück, wenn der Prozeß *p* beendet ist. War der Prozeß bereits beendet, so kehrt *Wait* sofort zurück.

Das Ergebnis ist dann der Wert, den die Prozedur des Prozesses zurückgegeben hat. Um hier beliebige Werte speichern zu können, muß eine Erweiterung von *BasicTypes.ANYDesc* definiert werden, die die entsprechenden Daten enthält. Über einen Typeguard kann dann auf diese Daten zugegriffen werden.

Wurde der Prozeß durch einen Aufruf von *HALT* oder einen Laufzeitfehler beendet, so ist das Ergebnis *NIL*.

Wait kann auch von anderen Prozessen als demjenigen, der *p* gestartet hat, aufgerufen werden.

PROCEDURE (p: Process) isRunning(): BOOLEAN;

isRunning prüft, ob der Prozeß *p* bereits beendet ist. Ist dies der Fall, wird *FALSE* zurückgegeben. Läuft *p* dagegen noch, so ist das Ergebnis *TRUE*.

Liefert *isRunning* den Wert *FALSE*, so kann das Ergebnis des Prozesses mit *Wait* bestimmt werden. *Wait* wartet dann nicht, sondern kehrt sofort zurück.

Beispielprogramm: Das folgende Programm startet die Prozedur *New* als neuen Prozeß. Der neue Prozeß öffnet ein Fenster mit dem Modul *Display*. Über den Parameter *w* wird dem neuen Prozeß die Größe dieses Fensters mitgeteilt. In das Fenster malt der Prozeß eine einfache Grafik.

Sobald der Benutzer 'S' drückt, startet *New* einen weiteren Unterprozeß. Dieser erhält einen neuen Parameter *w*, der die Koordi-

naten eines gegenüber dem letzten nach rechts unten verschobenen Fensters enthält. So können beliebig viele Unterprozesse und Unter-Unterprozesse gestartet werden. Durch das Schließsymbol können diese Prozesse einzeln wieder beendet werden.

Da die CLOSE-Anweisung des Moduls *Display* erst dann ausgeführt werden darf, wenn alle Prozesse beendet wurden, muß in der CLOSE-Anweisung dieses Moduls auf das Ende aller Prozesse gewartet werden.

Der Quelltext:

```
MODULE ConcurrencyDemo;

IMPORT
  C  := Concurrency,
  BT := BasicTypes,
  E  := Exec,
  I  := Intuition,
  D  := Display;

VAR p: C.Process;

TYPE
  Window = POINTER TO WindowDesc;
  WindowDesc = RECORD (BT.ANYDesc)
    x,y,w,h: INTEGER;
  END;

VAR w: Window;

PROCEDURE New(w: BT.ANY): BT.ANY;

VAR
  win: D.WindowPtr;
  i,top: INTEGER;
  msg: E.MessagePtr;
  fertig: BOOLEAN;
```

```
  PROCEDURE NewProcess;
  VAR
    new: Window;
  BEGIN
    NEW(new);
    new^ := w(Window)^;
    INC(new.y,20);
    INC(new.x,20);
    IF C.NewProcess(New,new)=NIL THEN END;
  END NewProcess;

BEGIN

  WITH w: Window DO

    NEW(win);
    IF D.OpenWindow(win,"Neuer Prozeß",
                    w.x,w.y,w.w,w.h,NIL) THEN
      I.OldModifyIDCMP(win.window,
          LONGSET{I.closeWindow,I.newSize,
                  I.vanillaKey});

      top := win.rp.font.ySize;
      fertig := FALSE;

      REPEAT
        D.FrontPen(win,1);
        D.Jam1(win);
        D.Home(win);
        D.Write(win,
           "Taste <S> startet noch einen Prozeß!");
        D.Complement(win);
        msg := NIL;
        WHILE msg=NIL DO
          i := 0;
          WHILE (i<=win.width-2) & (msg=NIL) DO
            D.Line(win,0             ,top          ,
                       win.width-1-i,win.height-1);
            D.Line(win,win.width-1-i,top          ,
                       win.width-1  ,win.height-1);
```

```
          msg := E.GetMsg(win.window.userPort);
          INC(i,2);
        END;
      END;

      IF msg#NIL THEN
        WITH msg: I.IntuiMessage DO
          IF    I.newSize     IN msg.class THEN
            D.Init(win); D.Clear(win);
          ELSIF I.vanillaKey  IN msg.class THEN
            IF CAP(CHR(msg.code))="S" THEN
              NewProcess
            END;
          ELSIF I.closeWindow IN msg.class THEN
            fertig := TRUE;
          END;
          E.ReplyMsg(msg);
        END;
      END;

    UNTIL fertig;

    D.Close(win);
  END;    (* IF D.OpenWindow(..) END; *)
 END;    (* WITH w: Window DO *)
 RETURN NIL;
END New;

BEGIN

 NEW(w);
 w.x := 0;   w.y := 0;
 w.w := 320; w.h := 70;
 p := C.NewProcess(New,w);

CLOSE

 C.WaitForAllProcesses;

END ConcurrencyDemo.
```

25. Modulbibliothek: Amigaspezifische Module



Obwohl einige der bisher beschriebenen Module auch speziell für den Amiga zugeschnittene Prozeduren enthielten, können sie ähnlich auch auf anderen Rechnern implementiert werden. Die in diesem Kapitel beschriebenen Module sind speziell zur Erleichterung der Programmierung einiger Amiga-Besonderheiten gedacht.

Alerts

```
DEFINITION Alerts;

IMPORT SYSTEM;

PROCEDURE Alert(s: ARRAY  OF CHAR;
                data..: SYSTEM.ADDRESS): BOOLEAN;

END Alerts.
```

Dieses Modul exportiert lediglich eine Prozedur:

**PROCEDURE Alert(s: ARRAY OF CHAR;
data..: SYSTEM.ADDRESS): BOOLEAN;**

Der Text *s* wird mit *RawDoFmt* aus Exec formatiert. Dabei enthält der Listenparameter *data* die Daten, die in die formatierte Zeichenkette eingefügt werden sollen. Dies funktioniert wie bei *io.Format*, siehe Kapitel 22 und [RKM: Libraries 92].

Die erzeugte Zeichenkette wird dann in eine Alert-Meldung umgewandelt und mit *DisplayAlert* aus *Intuition* ausgegeben. *Alert* kann somit zur Fehlersuche oder zum Anzeigen von schweren Fehlern verwendet werden.

Beispiel: In diesem Programm wird von der Möglichkeit gebrauch gemacht, in Amiga Oberon Zeichenkettenkonstanten über mehrere Zeilen zu verteilen.

```
MODULE AlertDemo;

IMPORT Alerts;

VAR
  text: POINTER TO ARRAY 80 OF CHAR;
  vier: LONGINT;

BEGIN
  NEW(text); text^ := "Alerts";
  vier := 4;
  REPEAT
  UNTIL ~ Alerts.Alert(
     "Dies ist ein mit %s erzeugter Alert!\n"
     "Er besteht aus %ld Zeilen, die von Alerts\n"
     "automatisch angordnet werden!\n"
     "Linke Taste = nochmal   Rechte Taste = Ende",
     text,vier);
END AlertDemo.
```

Die erzeugte Alert-Meldung ist:

```
Dies ist ein mit Alert erzeugter Alert!
Er besteht aus 4 Zeilen, die von Alerts
automatisch angordnet werden!
Linke Taste = nochmal   Rechte Taste = Ende
```

Beep

```
DEFINITION Beep;

PROCEDURE Beep(low: BOOLEAN);

END Beep.
```

Oft ist es sinnvoll, dem Benutzer eines Programmes über ein kurzes akustisches Signal über etwas zu Informieren, wie etwa über einen Fehler oder die Bereitschaft eines Programms nach einer längeren Berechnung neue Eingaben zu erhalten. Mit der folgenden Prozedur kann ein solches Signal erzeugt werden:

PROCEDURE Beep(low: BOOLEAN);

Über den Parameter *low* wird gewählt, ob ein hoher Signalton (*FALSE*) oder ein tieferer Warnton (*TRUE*) erzeugt werden soll.

Break

```
DEFINITION Break;

PROCEDURE CtrlCOff;
PROCEDURE CtrlCOn;
PROCEDURE CheckBreak;

END Break.
```

Das gewöhnliche Laufzeitsystem von Oberon-Programmen bietet keine Möglichkeit, diese mit der Tastenkombination Steuerung (Ctrl) und 'C' oder dem Shell-Befehl *break* abzubrechen (siehe Kapitel 15). Dies ist erst dann möglich, wenn das Modul *Break* von einem der Module des Programms importiert wird. Da die Abbruchsprüfung mit der Stackkontrolle gekoppelt ist, funktioniert dies jedoch nur, wenn mit eingeschalteter Stackkontrolle kompiliert wurde.

Beim Abbruch eines Programms mit Steuerung und 'C' gibt *Break* die Meldung ' *** *User Break* *** ' aus. Je nach dem, ob das Programm von einer Shell oder von der Workbench gestartet wurde, wird diese Meldung in einem Dialogfenster oder im Ausgabefenster der Shell ausgegeben.

Zudem stellt *Break* folgende Prozeduren zur Verfügung, mit denen die Abbruchsprüfung beeinflußt werden kann:

PROCEDURE CtrlCOff;
PROCEDURE CtrlCOn;

Mit diesen Prozeduren kann die Abbruchsprüfung zeitweise aus (*CtrlCOff*) und danach wieder eingeschaltet werden (*CtrlCOn*). Es ist in manchen Situationen nötig sicherzustellen, daß das Programm nicht abgebrochen wird.

Die beiden Prozeduren können geschachtelt aufgerufen werden, so daß beispielsweise nach zweifachem Aufruf von *CtrlCOff* die Abbruchskontrolle erst nach zwei Aufrufen von *CtrlCOn* wieder aktiviert wird.

PROCEDURE CheckBreak;

Innerhalb von Programmstücken, die keine Aufrufe von Prozeduren mit Stackkontrolle enthalten, kann mit dieser Prozedur explizit ein Abbruch geprüft werden. Dies ist z.B. dann nötig, wenn man in einer Schleife lediglich Routinen des Betriebssystems aufruft, jedoch auch hier einen Programmabbruch ermöglichen möchte.

BreakRq

```
DEFINITION BreakRq;

PROCEDURE CtrlCOff;
PROCEDURE CtrlCOn;
PROCEDURE CheckBreak;

END BreakRq.
```

Dieses Modul entspricht in seiner Funktionsweise exakt dem Modul *Break* (siehe oben). Zum Anzeigen des Abbruchs wird hier jedoch auf jeden Fall ein Dialogfenster verwendet.

Icons

```
DEFINITION Icons;

PROCEDURE PutIcon(Iconname: ARRAY OF CHAR;
                        to: ARRAY OF CHAR);

END Icons.
```

Dieses Modul wird von den Programmen von Amiga Oberon verwendet, um die erzeugten Dateien mit Piktogrammen auszustatten. Die Prozedur, die diese Piktogramme erzeugt, ist

**PROCEDURE PutIcon(Iconname: ARRAY OF CHAR;
to: ARRAY OF CHAR);**

Die Datei mit dem Namen *to* wird mit dem Piktogramm *Iconname* aus *'OBERON:Icons'* versehen. *Iconname* und *to* müssen dabei ohne Endung *'.info'* angegeben werden. Hat die Zieldatei bereits ein Piktogramm, so wird kein neues Piktogramm erzeugt, sondern das alte unverändert gelassen.

Beispiel:

```
Icons.PutIcon("txt","RAM:Test.mod");
```

Die Datei *'RAM:Test.mod'* wird mit dem Piktogramm für Texte versehen.

NoGuru

```
DEFINITION NoGuru;

PROCEDURE Assert(cc: BOOLEAN;
                msg: ARRAY  OF CHAR);

END NoGuru.
```

Wie in Kapitel 15 beschrieben fängt das gewöhnliche Laufzeitsystem für Amiga Oberon Laufzeitfehler zwar ab, gibt dabei jedoch keine sinnvollen Fehlermeldungen aus. Erst wenn eines der Module *NoGuru* und *NoGuruRq* importiert wird, werden solche Meldungen erzeugt. Es sind dabei folgende Laufzeitfehler möglich:

```
Busfehler
Addressfehler
Illegaler Befehl
Division durch 0
Bereichsfehler (CHK)
Überlauf (TRAPV)
Privilegverletzung
Trace-Vektor
Line-A
Line-F
Trap # 0 (Bereichsfehler)
Trap # 1 (ungültiger CASE-Index)
Trap # 2 (Stack überlauf)
Trap # 3 (Nil-Zeiger dereferenziert)
Trap # 4 (Funktion ohne RETURN beendet)
Trap # 5 (Typ-Check Fehler)
Trap # 6 (falscher Prozessor installiert)
Trap # 7 (Zeiger ist ungerade (Adressfehler))
Trap # 8 (Benutzerunterbrechung, ^C)
Trap # 9 (Speichermangel)
```

Die Fehlermeldungen *Busfehler*, *Adressfehler*, *Illegaler Befehl*, *Privilegverletzung*, *Trace-Vektor*, *Line-A* und *Line-F* weisen auf die entsprechenden Ausnahmesituationen des MC68000-Prozessors hin ([Williams 89[). Sie treten in gewöhnlichen Oberon-Programmen nicht auf, können jedoch durch grobe Fehler beim Arbeiten mit den Prozeduren aus *SYSTEM* und mit dem Betriebssystem auftreten.

Die anderen Meldungen sind gewöhnlich die entsprechenden Fehler im Oberon-Programm. Sie werden an der entsprechenden Ausnahmesituation des MC68000 erkannt, beispielsweise ein Typ-Check Fehler an einem *Trap #5*.

Außer der Fehlermeldung selbst gibt *NoGuru* noch die Inhalte der Register des Prozessors beim Auftreten des Fehlers aus. Dies wurde in Kapitel 15 genauer beschrieben.

NoGuru gibt die Fehlermeldungen mit Hilfe des Moduls *io* aus, so daß ein Programm, das *NoGuru* benutzt, beim Workbench-Start ein *io*-Fenster öffnet.

NoGuru exportiert nur eine einzige Prozedur:

PROCEDURE Assert(cc: BOOLEAN; msg: ARRAY OF CHAR);

> Wie die gleichnamige Prozedur aus *Requests* (Kapitel 22) kehrt diese Prozedur ohne etwas zu tun zurück, wenn der Parameter *cc* den Wert *TRUE* hat. Sonst wird die Meldung *msg* im *io*-Fenster ausgegeben und das Programm wird mit *HALT(20)* abgebrochen.

NoGuruRq

```
DEFINITION NoGuruRq;

END NoGuruRq.
```

Dieses Modul gibt Laufzeitfehler aus, wie es auch das Modul *NoGuru* tut. *NoGuruRq* gibt die Registerinhalte jedoch nicht aus. Außerdem verwendet *NoGuruRq* kein *io*-Fenster, sondern verwendet die Prozedur *Fail* aus *Requests* (Kapitel 22), so daß die Meldung in einem Dialogfenster oder im Shell-Fenster ausgegeben wird, je nach dem, ob das Programm von der Workbench oder von einer Shell aus gestartet wurde.

NoGuruRq bietet die Prozedur *Assert* nicht an. Stattdessen muß *Assert* aus Requests benutzt werden. Da *Requests* von *NoGuruRq* ohnehin importiert wird, wird dadurch das Programm nicht verlängert.

SecureDos

```
DEFINITION SecureDos;

IMPORT d := Dos;

VAR
  Me : d.ProcessPtr;
  oldCurrentDir : d.FileLockPtr;

PROCEDURE Open(name: ARRAY  OF CHAR;
                accessMode: LONGINT): d.FileHandlePtr;
PROCEDURE OpenFromLock(
                lock: d.FileLockPtr): d.FileHandlePtr;
PROCEDURE Close(file: d.FileHandlePtr);
PROCEDURE Lock(name: ARRAY  OF CHAR;
                accessMode: LONGINT): d.FileLockPtr;
PROCEDURE ParentDir(
                lock: d.FileLockPtr): d.FileLockPtr;
PROCEDURE DupLock(
                lock: d.FileLockPtr): d.FileLockPtr;
PROCEDURE CreateDir(
                name: ARRAY  OF CHAR): d.FileLockPtr;
PROCEDURE ParentOfFH(
                fh: d.FileHandlePtr): d.FileLockPtr;
PROCEDURE DupLockFromFH(
                fh: d.FileHandlePtr): d.FileLockPtr;
PROCEDURE UnLock(lock: d.FileLockPtr);

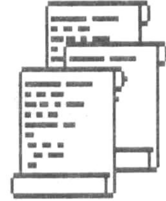
END SecureDos.
```

Die Dos-Library des Amiga-Betriebssystems merkt sich leider nicht, welches Programme welche Dateien geöffnet und welche Locks erstellt hat. Jedes Programm muß hierüber selbst Buch führen und alle Dateien schließen und Locks freigeben, bevor es beendet wird. Dies führt zu recht aufwendigen CLOSE-Anweisungsteilen, vor allem dann, wenn ein Programm jederzeit durch einen Fehler oder einen Benutzerabbruch mit <Steuerung> und <C> beendet werden kann.

Diese Arbeit wird von dem Modul *SecureDos* übernommen. Das Modul bietet Prozeduren der Dos-Library an, die Dateien öffnen und schließen oder Locks erzeugen und wieder freigeben. Die Prozeduren werden genauso aufgerufen wie ihre Gegenstücke aus der Dos-Library. Der Unterschied ist jedoch, daß diese Prozeduren über die geöffneten Dateien und erstellten Locks Buch führen, so daß bei einem Programmabbruch im CLOSE-Anweisungteil von *SecureDos* alle noch offenen Dateien geschlossen und alle erstellten Locks freigegeben werden.

Damit *SecureDos* funktionieren kann, dürfen die Prozeduren aus *SecureDos* nicht gemischt mit denen aus *Dos* aufgerufen werden. So muß beispielsweise eine mit *SecureDos.Open* geöffnete Datei auch mit *SecureDos.Close* geschlossen werden und darf keinesfalls mit *Dos.Close* geschlossen werden.

26. Modulbibliothek: Oberon-Unterstützung



Die in diesem Abschnitt beschriebenen Module enthalten Prozeduren, die Amiga-Oberon-Programme zu ihrer Ausführung benötigen. Die Module werden bei Bedarf automatisch vom Compiler eingebunden. Sie sind für gewöhnliche Oberon-Programmierer nicht von Bedeutung. Wer sich jedoch mit den Innereien und der Funktionsweise der mit Amiga Oberon übersetzten Programme beschäftigen möchte, kann aus diesen Modulen viele Informationen gewinnen.

Debug

```
DEFINITION Debug;

IMPORT e := Exec;

TYPE
  String = ARRAY 256 OF CHAR;
  StringPtr = UNTRACED POINTER TO String;

VAR
  Module : StringPtr;

PROCEDURE Trace(stat: INTEGER);
PROCEDURE NewProc(vars: e.ADDRESS; proc: INTEGER);
PROCEDURE EndProc;
PROCEDURE NewMod(mod: INTEGER; vars: e.ADDRESS);
PROCEDURE VarBase(vars: e.ADDRESS);

END Debug.
```

Bei der Compilation mit der Option '*-g*' bzw. mit dem Piktogramm-Merkmal *DEBUG=TRUE* werden die Prozeduren dieses Moduls verwendet, um die Kommunikation zwischen dem debuggten Programm und dem Debugger ODebug (als Zusatzpaket erhältlich, siehe

Kapitel 28) zu ermöglichen. Diese Kommunikation geschieht über gewöhnliche Exec-Messages.

GarbageCollector

```
DEFINITION GarbageCollector;
IMPORT Exec, SYSTEM;
CONST
  many = MAX(LONGINT) DIV 8;
TYPE
  Big = ARRAY many OF SYSTEM.ADDRESS;
  ObjectTypePtr = UNTRACED POINTER TO ObjectType;
  ObjectType = STRUCT
    id : LONGINT;
  END;
CONST
  usualObject = 1;
TYPE
  UsualObjectTypePtr =
    UNTRACED POINTER TO UsualObjectType;
  UsualObjectType = STRUCT (bt : ObjectType)
    size : LONGINT;
    numRefs : LONGINT;
    refs : Big;
  END;
CONST
  openArrayObject = 2;
TYPE
  OpenArrayObjectType = STRUCT (bt : ObjectType)
    size : LONGINT;
    numRefs : LONGINT;
    refs : Big;
  END;
  MutatorPtr = UNTRACED POINTER TO Mutator;
  InternalObjectTypePtr =
    UNTRACED POINTER TO InternalObjectType;
  InternalObjectType = STRUCT
    bt : ObjectType;
  END;
  VarsPtr = UNTRACED POINTER TO Vars;
```

```
  Vars = STRUCT
    next : VarsPtr;
    typ : UsualObjectTypePtr;
  END;
  Mutator = STRUCT
    globals : VarsPtr;
    locals : VarsPtr;
  END;
  ObjectPtr = UNTRACED POINTER TO Object;
  Object = STRUCT
    next : ObjectPtr;
    flags : SET;
    shade : SET;
    typ : InternalObjectTypePtr;
    mem : Big;
  END;
  OpenArrayObjectPtr =
    UNTRACED POINTER TO OpenArrayObject;
  OpenArrayObject = STRUCT
    length : LONGINT;
    object : Object;
  END;
CONST
  shadeOffset = -6;
  byteGrayOffset = -5;
  gray = 0;
  pink = 2;
  MaxMutators = 3FFFH;
TYPE
  GarbageCollectorBasePtr =
    UNTRACED POINTER TO GarbageCollectorBase;
  GarbageCollectorBase = STRUCT
    (libNode : Exec.Library)
    activeObjects : LONGINT;
    deadObjects : LONGINT;
    totalMem : LONGINT;
    cycleCount : LONGINT;
  END;
VAR
  base : GarbageCollectorBasePtr;
```

```
  mutator : Mutator;
  mutatorValid : BOOLEAN;
PROCEDURE AddGlobals(VAR mutator: Mutator;
                  adr: VarsPtr);
PROCEDURE RemGlobals(VAR mutator: Mutator);
PROCEDURE AddLocals(VAR mutator: Mutator;
                  adr: VarsPtr);
PROCEDURE RemLocals(VAR mutator: Mutator);
PROCEDURE RemAllLocals(VAR mutator: Mutator);
PROCEDURE AddGlobalsLib(VAR mutator: Mutator;
                  adr: VarsPtr);
PROCEDURE RemGlobalsLib(VAR mutator: Mutator);
PROCEDURE AddLocalsLib(VAR mutator: Mutator;
                  adr: VarsPtr);
PROCEDURE RemLocalsLib(VAR mutator: Mutator);
PROCEDURE RemAllLocalsLib(VAR mutator: Mutator);
PROCEDURE AddMutator(
                  VAR mutator: Mutator): BOOLEAN;
PROCEDURE RemMutator(VAR mutator: Mutator);
PROCEDURE AllocType(VAR mutator: Mutator;
       typ: ObjectTypePtr): InternalObjectTypePtr;
PROCEDURE Alloc(typ: InternalObjectTypePtr;
                  VAR adr: SYSTEM.ADDRESS);
PROCEDURE NewPreferences;
PROCEDURE AllocOpenArray(
                  typ: InternalObjectTypePtr;
                  VAR adr: SYSTEM.ADDRESS;
                  length: LONGINT);
PROCEDURE WaitForCollector(id: MutatorPtr);
PROCEDURE AllocFlag(): LONGINT;
PROCEDURE FreeFlag(flag: INTEGER);
PROCEDURE AssignRef(ref: SYSTEM.ADDRESS;
                  to: SYSTEM.ADDRESS);
PROCEDURE Assign(from, to: SYSTEM.ADDRESS;
                  type: ObjectTypePtr);
PROCEDURE AssignOpenArray(from,
                  to: SYSTEM.ADDRESS;
                  elementType: UsualObjectTypePtr;
                  length: LONGINT);
PROCEDURE AssignRefs(from, to: SYSTEM.ADDRESS;
```

```
                     type: ObjectTypePtr);
PROCEDURE CallAssign(from, to: SYSTEM.ADDRESS;
                     type: ObjectTypePtr);
PROCEDURE CallAssignOpenArray(
                     from, to: SYSTEM.ADDRESS;
                     elementType: UsualObjectTypePtr;
                     length: LONGINT);
PROCEDURE AssignRecord(from, to: SYSTEM.ADDRESS;
                     type: UsualObjectTypePtr);
PROCEDURE New(VAR adr: SYSTEM.ADDRESS;
                     typ: InternalObjectTypePtr);
PROCEDURE NewOpenArray(VAR adr: SYSTEM.ADDRESS;
                     typ: InternalObjectTypePtr;
                     size: LONGINT);
PROCEDURE Allocate(VAR adr: SYSTEM.ADDRESS;
                     typ: InternalObjectTypePtr);
PROCEDURE AllocateOpenArray(
                     VAR adr: SYSTEM.ADDRESS;
                     typ: InternalObjectTypePtr;
                     size: LONGINT);
PROCEDURE DuplicateOpenArray(VAR from, to:
                     ARRAY 100000H OF SYSTEM.ADDRESS;
                     typ: InternalObjectTypePtr;
                     elementTyp: UsualObjectTypePtr;
                     dims: INTEGER);
PROCEDURE AddType(VAR to: SYSTEM.ADDRESS;
                     typ: ObjectTypePtr);

END GarbageCollector.
```

Dieses Modul und die Funktionsweise des Garbage-Collectors hier vollkommen zu erklären würde den Rahmen dieses Handbuchs sprengen. Genauere Informationen hierzu sind unter Angabe von (überzeugenden) Gründen vom Autor zu erhalten.

Hier sollen nur ein paar Hinweise zu einzelnen Typen und Routinen gegeben werden. Für jeden Mutator muß ein Objekt des Typs *Mutator* erzeugt werden, das dem Garbage-Collector mit *AddMutator* überge-

ben wird. Dieses Objekt enthält Listen *globals* und *locals*, die die globalen und lokalen Variablen des Mutators beschrieben.

Die wichtigsten Prozeduren von *GarbageCollector* sind:

AddGlobals, RemGlobals, AddLocals, RemLocals

Diese Prozeduren informieren den Garbage-Collector über einen neuen Bereich globaler bzw. lokaler Variablen oder entfernen den zuletzt hinzugefügten Bereich. Für eine bessere Effizient rufen Oberon-Programme diese Routinen nicht auf, stattdessen erzeugt der Compiler direkt Code, der diesen Prozeduren entspricht.

PROCEDURE AddMutator(VAR mutator: Mutator
): BOOLEAN;
PROCEDURE RemMutator(VAR mutator: Mutator);

Mit diesen Prozeduren wird der Garbage-Collector über neue Mutatoren unterrichtet bzw. entfernt diese wieder. Diese Prozeduren werden z.B. von dem Modul *Concurrency* (Kapitel 24) verwendet, damit die neu gestarteten Prozesse auch mit verfolgten Objekten arbeiten können.

PROCEDURE NewPreferences;

Diese Prozedur verwendet GarbagePrefs (Kapitel 16) dazu, der Garbage-Collector-Library mitzuteilen, daß sich die Preferences geändert haben. Sobald die Library von keinem Programm mehr geöffnet ist werden die Voreinstellungen aktiv.

PROCEDURE WaitForCollector(id: MutatorPtr);

Diese Prozedur kehrt erst dann zurück, wenn der Collector seinen aktuellen Zyklus beendet hat. Währenddessen wird die Priorität des Collectors auf die des Tasks gesetzt, der *WaitForCollec-*

tor aufgerufen hat.

Um sicherzustellen, daß alle unerreichbaren Objekte eines Mutators freigegeben werden, muß *WaitForCollector* zweimal hintereinander aufgerufen werden.

**PROCEDURE AssignRef(ref: SYSTEM.ADDRESS;
to: SYSTEM.ADDRESS);**

Diese Prozedur kann zum Zuweisen von verfolgten Zeigern verwendet werden. Dabei muß *ref* ein verfolgter Zeiger und *to* die Adresse einer verfolgten Variablen sein.

Oberon-Programme benutzen diese Prozedur nicht für Zeigerzuweisungen, da dies zu ineffizient wäre. Diese Prozedur kann jedoch z.B. in Assembler- oder C-Unterroutinen verwendet werden, wenn eine Zuweisung von verfolgten Zeigervariablen nötig ist. Es ist hier jedoch wichtig, daß *to* die Adresse einer verfolgten Variablen ist.

**PROCEDURE New(VAR adr: SYSTEM.ADDRESS;
typ: InternalObjectTypePtr);
PROCEDURE NewOpenArray(VAR adr: SYSTEM.ADDRESS;
typ: InternalObjectTypePtr;
size: LONGINT);**

Diese Prozeduren werden automatisch aufgerufen, wenn mit *NEW* Speicher für einen verfolgten Zeiger angefordert wird. Sonst sollten diese Prozeduren nicht aufgerufen werden. *New* wird zum Anfordern von gewöhnlichen Objekten, *NewOpenArray* zum Reservieren von Speicher für offene Feldvariablen verwendet.

Bei Speichermangel rufen diese Prozeduren die Routine *OberonLib.OutOfMemHandler* auf (siehe unten), so daß eine in *OberonLib* installierte Prozedur zum Abfangen von Speicher-

mangel auch beim Anfordern von Speicher vom Garbage-Collector verwendet wird.

PROCEDURE Allocate(VAR adr: SYSTEM.ADDRESS;
typ: InternalObjectTypePtr);
PROCEDURE AllocateOpenArray(
VAR adr: SYSTEM.ADDRESS;
typ: InternalObjectTypePtr;
size: LONGINT);

Diese beiden Prozeduren entsprechen den oben beschriebenen Prozeduren *New* und *NewOpenArray*. Sie rufen jedoch *OberonLib.OutOfMemHandler* nicht auf, sondern liefern bei Speichermangel *NIL*. Der Compiler benutzt diese Prozeduren, wenn mit *SYSTEM.ALLOCATE* Speicher angefordert wird.

OberonLib

```
DEFINITION OberonLib;

IMPORT Exec, SYSTEM;

TYPE
  Proc = UNTRACED POINTER TO STRUCT END;
  PROC = PROCEDURE;
  TaskTrapDataPtr =
    UNTRACED POINTER TO TaskTrapData;
  TaskTrapData = STRUCT
    mutator : SYSTEM.ADDRESS;
    a5 : SYSTEM.ADDRESS;
    haltProc : PROC;
    oldSP : SYSTEM.ADDRESS;
    user : ARRAY 32 OF SYSTEM.ADDRESS;
  END;

VAR
  wbStarted : BOOLEAN;
  dosCmdLen : LONGINT;
```

```
  dosCmdBuf : Exec.APTR;
  wbenchMsg : Exec.MessagePtr;
  closing : BOOLEAN;
  Break : BOOLEAN;
  HaltProc : PROC;
  Result : LONGINT;
  OldSP : UNTRACED POINTER TO STRUCT
    returnAdr : LONGINT;
    stackSize : LONGINT;
  END;
  Me : Proc;
  MemReqs : LONGSET;
  OutOfMemHandler : PROCEDURE();
  execBase : UNTRACED POINTER TO STRUCT
    thisTask : UNTRACED POINTER TO STRUCT
      trapData : TaskTrapDataPtr;
    END;
  END;

PROCEDURE Mul(a, b: LONGINT): LONGINT;
PROCEDURE ModDiv(a, b: LONGINT): LONGINT;
PROCEDURE New(VAR adr: Exec.APTR; size: LONGINT);
PROCEDURE Allocate(VAR adr: Exec.APTR;
                   size: LONGINT);
PROCEDURE Dispose(VAR adr: Exec.APTR);
PROCEDURE Copy(source: ARRAY OF CHAR;
               VAR dest: ARRAY OF CHAR);
PROCEDURE StackChk(size: LONGINT);
PROCEDURE CheckProcessor(what: SET);
PROCEDURE SetA5;
PROCEDURE AllocUser(): INTEGER;
PROCEDURE FreeUser(i: INTEGER);

END OberonLib.
```

OberonLib ist das Basismodul aller mit Amiga Oberon übersetzten Programme. Es wird automatisch beim Compilieren eines Moduls importiert. Der BEGIN-Anweisungsteil von *OberonLib* wird damit immer als erster ausgeführt, der CLOSE-Anweisungsteil entsprechend

als letzter. *OberonLib* tut alles beim Programmstart nötige. Beim Start von der Shell speichert es die Argumentzeile ([AmigaDos 91]), beim Start von der Workbench wird die Workbenchmessage geholt und im CLOSE-Anweisungsteil beantwortet ([RKM: Libraries 92]).

OberonLib verwaltet den Speicher, der für nicht verfolgte Objekte angefordert wird und gibt diesen automatisch beim Programmende frei. *OberonLib* enthält grundlegende Prozeduren zum Rechnen mit *LONGINT*-Zahlen und zum Kopieren von Zeichenketten. Die Stackkontrolle und die Überprüfung, ob ein für einen speziellen Prozessor optimiertes Programm auch auf einem Rechner mit diesem Prozessor gestartet wurde, werden auch von *OberonLib* übernommen.

Mit bedingter Compilation kann aus *OberonLib* die für das Linken von Libraries und Devices nötige Datei *LibOberonLib.obj[s][a]* erzeugt werden (siehe Kapitel 11). Dazu müssen folgende Anweisungen in eine Shell eingegeben werden:

```
> OBERON SET LibLink OberonLib.mod
> RENAME obj/OberonLib.obj AS obj/LibOberonLib.obj
```

OberonLib setzt das Element *trapData* der Task-Struktur des Programms auf einen Zeiger auf eine Struktur vom Typ *TaskTrapData*. In dieser sind für das Programm wichtige Daten enthalten:

TaskTrapData.mutator

Bei Verwendung des Garbage-Collectors zeigt dieses Feld auf die *Mutator*-Struktur des Prozesses. Beim Starten von neuen Prozessen mit den Prozeduren aus *Concurrency* (Kapitel 24) wird dieses Feld auch bei neuen Prozessen korrekt gesetzt.

TaskTrapData.a5

Dieser Wert wird nur bei Verwendung des kleinen Datenmodells

benötigt. Es ist ein Zeiger auf den Bereich der globalen Variablen des Programms.

TaskTrapData.haltProc

Diese Prozedur wird beim Aufruf der Standardprozedur *HALT* von *HaltProc* (s.u.) aufgerufen. *Concurrency* setzt hier eine Prozedur ein, die den betroffenen Prozeß beendet.

TaskTrapData.oldSP

Dies ist der Wert des Prozessorregisters A7 beim Start des Prozesses bzw. des Programms.

TaskTrapData.user

Dieses Feld kann von den Modulen eines Oberon-Programms für eigene Werte verwendet werden, so benutzt *Concurrency* dieses Feld, um für jeden Prozeß einen Zeiger auf die entsprechende Process-Struktur zu speichern. Auf ein Element dieses Feldes darf nur zugegriffen werden, wenn es zuvor mit *AllocUser* (siehe unten) angefordert wurde.

OberonLib exportiert folgende Variablen:

wbStarted

Diese Variable ist *TRUE*, wenn das Programm von der Workbench gestartet wurde, sonst *FALSE*.

dosCmdLen, dosCmdBuf

Diese beiden Variablen sind nur definiert, wenn das Programm von einer Shell gestartet wurde, also wenn *~wbStarted* erfüllt ist. Dann enthalten sie die Länge (*dosCmdLen*) und die Speicheradresse (*dosCmdBuf*) der Argumentzeile (siehe [AmigaDos 91]).

Auf diese Variablen sollte jedoch nicht direkt zugegriffen werden, dies geschieht komfortabler mit den Prozeduren des Moduls *Arguments* (Kapitel 22).

wbenchMsg

Dies Variable ist nur definiert, wenn das Programm von der Workbench gestartet wurde, also wenn *wbStarted* den Wert *TRUE* hat. Dann ist dies ein Zeiger auf die von der Workbench erhaltene Nachricht (siehe [RKM: Libraries 92]). Mit ihr kann auf die Workbenchargumente zugegriffen werden. Dies geschieht jedoch komfortabler über das Modul *Arguments* (Kapitel 22).

closing

Diese Variable ist *TRUE* während die CLOSE-Anweisungsteile des Programms ausgeführt werden, sonst ist sie *FALSE*.

Break

Break wird von den Modulen *Break* und *BreakRq* auf *TRUE* gesetzt, wenn das Programm abgebrochen wurde. Die Prozedur *StackChk* (siehe unten) sorgt dann für einen Programmabbruch, wenn sie aufgerufen wird.

HaltProc

Diese Prozedur wird aufgerufen, wenn der Standardbefehl *HALT* ausgeführt wird.

Result

Diese Variable enthält den Rückgabewert des Programms. *HALT(n)* entspricht den beiden Anweisungen

```
OberonLib.Result := n;
OberonLib.HaltProc;
```

OldSP

Diese Variable enthält den Wert des Stackpointers (Prozessorregister A7) beim Programmstart. *OldSP.stackSize* enthält die Größe des Stapelspeichers des Hauptprozesses des Programms.

Me

Diese Variable zeigt auf die Prozeß-Struktur des Hauptprozesses des Programms. Da *OberonLib* keine Module importieren darf, die nicht implementationslos sind, also auch nicht *Dos*, ist der Typ dieser Variablen nicht *Dos.ProcessPtr*.

MemReqs

Diese Variable legt die Art des Speichers, der für nicht verfolgte Zeigervariablen angefordert werden soll. Dieser Wert wird direkt an *Exec.AllocMem* übergeben. Er enthält gewöhnlich den Wert *LONGSET{Exec.memClear}*. Um Grafikspeicher zu allozieren, ist folgende Anweisung nötig:
Es ist hier wichtig, das die Variable *bitPlanePtr* kein verfolgter Zeiger ist, als etwa *UNTRACED POINTER TO ARRAY n OF INTEGER*. Für verfolgte Objekte kann der Speichertyp nicht angegeben werden.

```
INCL(OberonLib.MemReqs,Exec.chip);
NEW(bitPlanePtr)
EXCL(OberonLib.MemReqs,Exec.chip);
```

OutOfMemHandler

Diese Prozedurvariable kann auf eine Prozedur gesetzt werden, die aufgerufen wird, wenn *NEW* keinen Speicher anfordern kann. In dieser Prozedur kann Speicher freigegeben oder der Benutzer auf den Speichermangel aufmerksam gemacht werden. Die Prozedur sollte nur dann zurückkehren, wenn für mehr freien Speicher gesorgt wurde. Dann wird erneut versucht, Speicher anzufordern. Schlägt auch dieser Versuch fehl, so wird *OutOfMemHandler* nochmals aufgerufen.

Gibt der *OutOfMemHandler* keinen Speicher frei, so besteht hier die Gefahr einer Endlosschleife. Daher darf die eingetragene Prozedur nur dann zurückkehren, wenn sie Speicher freigibt oder dem Benutzer eine Abbruchmöglichkeit bietet.

OberonLib setzt *OutOfMemHandler* auf eine Prozedur die lediglich einen Laufzeitfehler wegen Speichermangels verursacht. Das Modul *Requests* setzt *OutOfMemHandler* auf eine Prozedur, die in einem Dialogfenster den Benutzer über den Speichermangel informiert. Der Benutzter kann dabei wählen, ob das Programm abgebrochen, oder ob die Speicheranforderung nochmals versucht werden soll.

Ein Beispiel für einen *OutOfMemHandler* stellt folgende Prozedur dar:

```
PROCEDURE MemHandler;
VAR n: Lists.NodePtr;
BEGIN
  n := Lists.RemHead(list);
  IF (n=NIL) &
     ~ Requests.Request("Großes Problem:",
                 "Ich hab keinen Speicher mehr!",
                 "Nochmal versuchen","Abbrechen")
  THEN HALT(20) END;
END MemHandler;
```

In diesem Beispiel wird davon ausgegangen, daß mit Garbage-Collector compiliert wird, so daß das Entfernen eines Elementes aus der Liste genügt, um seinen Speicher freizugeben. Es wird hier also bei jedem Aufruf von *MemHandler* entweder ein Element der Liste entfernt und damit sein Speicher freigegeben, oder vom Benutzer die Bestätigung geholt, daß die Speicheranforderung nochmals ausgeführt werden soll. Hier wird davon ausgegangen, daß die Elemente der Liste relativ groß sind, so daß durch das Freigeben eines Elementes auch entscheidend Speicher freigegeben wird. Ansonsten sollten bei jedem Aufruf von *MemHandler* gleich mehrere Listenelemente freigegeben werden, da es sonst sehr lange dauern kann, bis genügend Speicher freigegeben wurde.

execBase

Dies ist im Grunde dieselbe Variable wie *Exec.exec*. Sie ist hier jedoch so definiert, daß sie nur das Recordelement *thisTask* enthält, das wiederum lediglich *trapData* enthält. *trapData* ist vom Typ *TaskTrapDataPtr*. Auf diese Weise kann man leicht auf das *TaskTrapData*-Feld (siehe oben) des aktiven Prozesses zugreifen. So greift man auf ein mit *AllocUser* angefordertes Feld beispielsweise mit folgendem Ausdruck zu:

```
OberonLib.execBase.thisTask.trapData.user[i]
```

Innerhalb einer mit *LibLink* erzeugten Library oder einem Device dürfen nur die Variablen *closing* und *MemReqs* verwendet werden. Alle anderen Variablen enthalten undefinierte Werte.

Im folgenden werden die von *OberonLib* exportierten Prozeduren beschrieben. Für viele dieser Prozeduren erzeugt der Oberon-Compiler automatisch Aufrufe, da sie verschiedenen Operationen der Sprache Oberon entsprechen.

PROCEDURE Mul(a, b: LONGINT): LONGINT;

Diese Prozedur wird bei der Multiplikation zweier *LONGINT*-Zahlen verwendet, wenn kein Code für einen MC68020-Prozessor erzeugt wird. Das Ergebnis ist das Produkt von *a*b*.

PROCEDURE ModDiv(a, b: LONGINT): LONGINT;

Diese Prozedur wird bei der Berechnung der Ganzzahldivision und des Modulos zweier *LONGINT*-Zahlen verwendet, wenn kein Code für einen MC68020-Prozessor erzeugt wird. Das Ergebnis ist das Produkt von *a DIV b* wird dabei wie üblich im Prozessorregister *D0* zurückgegeben. *D1* enthält nach dem Aufruf den Wert von *a MOD b*.

PROCEDURE New(VAR adr: Exec.APTR;
size: LONGINT);
PROCEDURE Dispose(VAR adr: Exec.APTR);
PROCEDURE Allocate(VAR adr: Exec.APTR;
size: LONGINT);

Diese Prozeduren werden aufgerufen, wenn eine der Standardprozeduren *NEW*, *DISPOSE* und *SYSTEM.ALLOCATE* beim Arbeiten mit nicht verfolgten Zeigern verwendet wird. *size* gibt dabei die Größe des anzufordernden Objektes in Bytes an. *New* ruft die Prozedur in der Variablen *OutOfMemHandler* auf, wenn der Speicher nicht alloziert werden konnte. Danach versucht *New* erneut, das Objekt zu allozieren.

PROCEDURE Copy(source: ARRAY OF CHAR;
VAR dest: ARRAY OF CHAR);

Diese Prozedur wird für die Standardprozedur *COPY* benutzt. Sie macht exakt dasselbe wie *COPY*, sie kopiert nämlich die Zeichenkette *source* in die Zeichenkette *dest*.

PROCEDURE StackChk(size: LONGINT);

Diese Prozedur wird benutzt, um zu prüfen, ob der Stapelspeicher noch genügend Platz für *size* Bytes bietet. Ist dies nicht der Fall, so kehrt *StackChk* nicht zurück, sondern löst einen Laufzeitfehler aus.

Zusätzlich prüft *StackChk*, ob die Variable *Break* den Wert *TRUE* enthält und bricht dann das Programm ab. Die Variable *Break* wird von den Modulen *Break* und *BreakRq* gesetzt, wenn das Programm abgebrochen wird.

PROCEDURE CheckProcessor(what: SET);

Module, die für einen speziellen Prozessor optimiert compiliert wurden, prüfen mit dieser Prozedur, ob der gewünschte Prozessor auch vorhanden ist. *CheckProcessor* bricht das Programm mit einem Laufzeitfehler ab, wenn dies nicht der Fall ist.

PROCEDURE SetA5;

Beim kleinen Datenmodell ist es notwendig, daß das Prozessorregister A5 immer einen Zeiger auf den Bereich der globalen Variablen enthält. In Prozeduren, in denen dies nicht der Fall ist, kann dieser Zeiger mit *SetA5* explizit geladen werden. Dies funktioniert jedoch nur dann, wenn der aktuelle Prozeß der Hauptprozeß des Oberon-Programms oder ein mit *Concurrency* gestarteter Prozeß ist.

So kann z.B. eine Prozedur, die als Interrupt installiert wird (siehe [RKM: Libraries 92]) A5 nicht mit *SetA5* laden. Hier muß ein anderer Weg gefunden werden, beispielsweise kann der Zeiger über den Parameter *data* übergeben werden.

Eine wichtige Anwendung von *SetA5* sind die Hook-Funktionen des AmigaOS, siehe dazu auch das nächste Kapitel.

PROCEDURE AllocUser(): INTEGER;
PROCEDURE FreeUser(i: INTEGER);

Mit diesen Prozeduren können die Elemente des *user*-Feldes in *TaskTrapData* angefordert und wieder freigegeben werden. Damit ein Modul auch mit anderen Modulen, die hier Einträge verwenden, zusammenarbeiten kann, müssen die Elemente von *user* mit diesen Prozeduren verteilt werden.

Sind alle Elemente bereits angefordert, so ist das Ergebnis von *AllocUser* der Wert *-1*.

Das Modul *Concurrency* z.B. benutzt diese Prozeduren, da es für jeden Prozeß einen Zeiger auf das Prozeßobjekt in *TaskTrapData.user* speichert.

OberonSupport

```
DEFINITION OberonSupport;

IMPORT
  e   := Exec,
  avl := UntracedAVL,
  rx  := Rexx;

TYPE
  Error = STRUCT
    num,line,column : INTEGER;
  END;
  ErrorHeaderPtr = UNTRACED POINTER TO ErrorHeader;
  ErrorHeader = STRUCT
    numErrors : INTEGER;
    error : ARRAY MAX(INTEGER) OF Error;
  END;

CONST
  name = "oberonsupport.library";
  version = 3;
```

```
VAR
  base : e.LibraryPtr;

PROCEDURE ReadErrorFile(
                 name: e.STRPTR): ErrorHeaderPtr;
PROCEDURE FreeErrorFile(errors: ErrorHeaderPtr);
PROCEDURE GetErrorText(num: INTEGER): e.STRPTR;
PROCEDURE MarkChanged(VAR name: avl.String);
PROCEDURE RemoveAllResidents;
PROCEDURE AllResident;
PROCEDURE AddResident(VAR name: avl.String);

END OberonSupport.
```

Die OberonSupport-Library ist für das Laden und Residenthalten der Symboldateien zuständig. Zudem stellt sie Routinen zum Auslesen der bei der Compilation aufgetretenen Fehlermeldungen zur Verfügung. Die öffentlich zugänglichen Prozeduren sind:

PROCEDURE ReadErrorFile(
name: e.STRPTR): ErrorHeaderPtr;

Die Fehlerdatei zum Quelltext *name* wird eingelesen. Das Ergebnis ist *NIL*, falls ein Fehler auftrat. Über die Elemente des zurückgegebenen *ErrorHeaders* können die Anzahl der Fehler und die Positionen und Nummern der einzelnen Fehler bestimmt werden.

PROCEDURE FreeErrorFile(errors: ErrorHeaderPtr);

Der Speicher der mit *ReadErrorFile* eingelesenen Fehlerdatei *errors* wird freigegeben.

PROCEDURE GetErrorText(num: INTEGER): e.STRPTR;

Das Ergebnis ist ein Zeiger auf die Fehlermeldung, die zu dem Fehler mit der Nummer *num* gehört.

PROCEDURE MarkChanged(VAR name: avl.String);

Die Symboldatei des Moduls *name* wird als verändert markiert. Ist diese Symboldatei speicherresident, so erzwingt dieser Aufruf das Neuladen der Datei. Diese Routine wird vom Compiler aufgerufen, wenn er eine neue Symboldatei erzeugt hat.

PROCEDURE RemoveAllResidents;

Die Liste der resident zu haltenden Symboldateien wird geleert. Danach wird keine Symboldatei mehr resident gehalten.

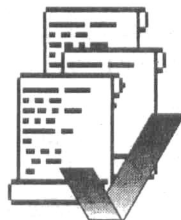
PROCEDURE AllResident;

Nach einem Aufruf dieser Routine werden alle Symboldateien resident gehalten.

PROCEDURE AddResident(VAR name: avl.String);

name wird in die Liste der resident zu haltenden Symboldateien aufgenommen. *name* ist dabei der Name des Moduls der gewünschten Symboldatei.

27. Benutzung der AmigaOS-Interface-Module



Die AmigaOS-Interface-Module ermöglichen den Zugriff auf die Routinen und Strukturen das Amiga-Betriebssystems von Oberon Programmen aus. Da diese Module sehr umfangreich sind, werden ihre Quelltexte oder Definitionsmodule hier nicht abgedruckt. Die Quelltexte sind jedoch im Verzeichnis 'Interfaces' auf den Disketten von Amiga Oberon enthalten.

Dieses Handbuch kann unmöglich eine komplette Anleitung zum Programmieren mit dem AmigaOS enthalten, dies würde sich über viele hundert Seiten erstrecken. Der Oberon-Programmierer sei hier auf auf die Amiga-Originalliteratur verwiesen: [RKM: Libraries 92], [RKM: Devices 91], [Style 91], [RKM: Hardware 91], [RKM: Autodocs 91] und [AmigaDos 91]. Hilfreich sind auch oft die Kommentare in den Quelltexten der AmigaOS-Interface-Module.

In diesem Kapitel sollen lediglich ein paar Beispiele beschrieben werden und es soll auf Besonderheiten von Amiga Oberon hingewiesen werden, die vor allem im Vergleich zur niedrigen Programmiersprache 'C', die auf dem Amiga weit verbreitet ist, deutlich werden. Die Spracherweiterungen, die den Zugriff auf das AmigaOS ermöglichen, wurden bereits in Kapitel 15 beschrieben.

Arbeiten mit Libraries

Die wichtigste Methode, um auf die Routinen des AmigaOS zuzugreifen, sind Funktionsbibliotheken, die sogenannten Libraries. Eine Library muß vor der Verwendung ihrer Routinen mit der Routine *OpenLibrary* der Exec-Library geöffnet und nach dem letzten Aufruf einer ihrer Routinen wieder geschlossen werden. Die einzige Ausnahme bildet hier die Exec-Library, die immer geöffnet ist.

Soll in einem Oberon-Modul eine Library verwendet werden, so muß

das Interface-Modul der Library importiert werden. Im BEGIN-Anweisungsteil des Interface-Moduls wird die Library automatisch geöffnet und im CLOSE-Anweisungsteil wird sie automatisch wieder geschlossen. So ist man von der lästigen Pflicht befreit, die Library selbst öffnen und beim Programmende korrekt schließen zu müssen.

In den Interface-Modulen werden die Routinen der Libraries als Oberon-Prozeduren definiert. Sie können wie ganz gewöhnliche Prozeduren aufgerufen werden. So zeigt das Programm

```
MODULE Sunday;
IMPORT Dos, io;
VAR
  date: Dos.Date;
BEGIN
  Dos.DateStamp(date);
  IF date.days MOD 7 = 0 THEN
    io.WriteString("Heute ist Sonntag!");
  ELSE
    io.WriteString("Heute ist nicht Sonntag!");
  END;
  io.WriteLn;
END Sunday.
```

an, ob es an einem Sonntag gestartet wurde. Der Aufruf *Dos.DateStamp(date)* springt dabei direkt die Routine der Dos-Library an.

Die Libraries, die es mindestens seit der AmigaOS-Version 1.2 gibt, werden mit der Versionsnummer 33 geöffnet. Schlägt das Öffnen fehl, so wird das Programm automatisch abgebrochen. Die Routinen von AmigaOS 1.2 können also ohne weitere Prüfungen verwendet werden. Da neuere Betriebssystemversionen jedoch entscheidende Verbesserungen und eine große Zahl an neuen Routinen mit sich gebracht haben, möchte man diese auch oft in Oberon-Programmen benutzen. Dann muß vor der Verwendung der Routinen der Library die Version entsprechend geprüft werden. Dazu exportiert jedes Library-Interface-Modul eine Variable, die die Basisadresse der Library enthält. Über

diese Variable kann die Version der Library geprüft werden. Ab welcher Libraryversion eine Routine zur Verfügung steht, kann den Quelltexten der Interface-Module und der Beschreibung der Routine in [RKM: Autodocs 91] entnommen werden. Ein Programm, das den Namen des aktuellen Wochentags ausgibt, sieht mit der Dos-Library Version 37 folgendermaßen aus:

```
MODULE Weekday;
IMPORT Dos, io;
VAR
  date: Dos.DateTime;
BEGIN
  IF Dos.dos.lib.version<37 THEN
    io.WriteString("Benötige dos.library V37");
  ELSE
    Dos.DateStamp(date);
    NEW(date.strDay);
    IF Dos.DateToStr(date) THEN
      io.WriteString("Heute ist ");
      io.WriteString(date.strDay^);
    ELSE
      io.WriteString("Fehler!");
    END;
  END;
  io.WriteLn;
END Weekday.
```

Die Interface-Module der Libraries, die in AmigaOS 1.2 noch nicht existierten, öffnen die Library zwar, sie prüfen jedoch nicht, ob das Öffnen erfolgreich war. So muß man hier explizit prüfen, ob die Basisadresse der Library ungleich *NIL* ist. So ist es jedoch möglich, Programme zu schreiben, die unter neueren Betriebssystemversionen von den neuen Fähigkeiten Gebrauch machen, unter älteren Versionen jedoch, evtl. eingeschränkt, dennoch funktionieren. Die Module, für die dies zutrifft, sind im folgenden aufgelisteten:

```
Commodities         GadTools            IFFParse
KeyMapLib           MathIEEESingBas MathIEEESingTrans
Utility             Workbench
```

Um beispielsweise zwei Zeichenketten ohne Berücksichtigung der Groß- und Kleinschreibung zu vergleichen, kann die Utility-Library des AmigaOS 2.0 verwendet werden. Die Vergleichsroutinen dieser Library berücksichtigen dabei auch die internationalen Zeichen korrekt:

```
MODULE Vergleich;
IMPORT Strings, Utility;
VAR
  a,b: ARRAY 80 OF CHAR;
  gleich: BOOLEAN;
  ...
BEGIN
  ...
   IF Utility.base#NIL THEN
     gleich := Utility.Stricmp(a,b)=0;
   ELSE
     Strings.Upper(a); Strings.Upper(b);
     gleich := a=b;
   END;
  ...
END Vergleich;
```

Unter älteren Versionen des AmigaOS funktioniert dieses Programm auch, bei internationalen Zeichen werden Groß- und Kleinbuchstaben dann jedoch als unterschiedlich angesehen.

Macht ein Modul massiv Gebrauch von den Funktionen einer Library, die nicht unter allen Versionen des AmigaOS zur Verfügung steht, so sollte das Modul zu Beginn prüfen, ob die nötige Version der Library vorhanden ist und gegebenenfalls das Programm abbrechen. Dies geschieht sehr einfach mit der Prozedur *Assert* von *Requests*:

```
Requests.Assert(GadTools.base#NIL,
                "Benötige gadtools.library V36!");
```

Die Strukturen des AmigaOS

Die von den Routinen des AmigaOS verwendeten Strukturen sind objektorientiert aufgebaut. So gibt es grundlegende Typen, wie den Knoten einer Liste *Node*, aus denen durch Erweiterung um neue Elemente komplexere Typen wie eine Nachricht *Message* oder durch weitere Spezialisierung eine Nachricht von Intuition *IntuiMessage* entsteht. Die Objekte enthalten jedoch leider keine für den Oberon-Compiler überprüfbare Typinformation, sie können daher nicht als Oberon-Records definiert werden. Stattdessen werden sie, wie in Kapitel 15 beschrieben wurde, mit dem erweiterterbaren Typ *STRUCT* nachgebildet.

Das AmigaOS kennt jedoch auch Typen, die getrennt definiert werden, die jedoch vom manchen Routinen trotz der unterschiedlichen Typen verwendet werden können. Ein Beispiel sind die Typen *MinNode* und *Node*, die als Parameter für die listenbearbeitenden Routinen der Exec-Library verwendet werden können. In Oberon könnte *MinNode* als Erweiterung von *Node* definiert werden, dabei wäre der Zugriff auf die Elemente dann jedoch anders als in der Sprache 'C' und damit anders als in der Literatur zur Programmierung des AmigaOS, was für viel Verwirrung sorgen würde.

Stattdessen wird im Exec-Interface-Modul ein leerer Grundtyp *CommonNode* definiert, von dem *MinNode* und *Node* erben:

```
CommonNodePtr = POINTER TO CommonNode;
CommonNode = STRUCT END;
Node          = STRUCT (dummy : CommonNode) ... END;
MinNode       = STRUCT (dummy : CommonNode) ... END;
```

Die Prozeduren, die mit Listen arbeiten und sowohl auf Knoten des Typs *MinNode* als auch auf *Node* angewendet werden können, benutzen den Typ *CommonNodePtr* als Parameter bzw. Ergebnistyp, der kompatibel zu beiden Knotentypen und allen ihren Erweiterungen ist.

Parameter der Libraryroutinen

Viele Routinen der Libraries erhalten Adressen von Variablen oder Daten als Parameter. In der Programmiersprache 'C' ist dies nicht problematisch, da diese Sprache einen Operator für die Bestimmung der Adresse enthält. Da eine solche Adreßarithmetik jedoch zu sehr komplexen und schwer zu findenden bösen Fehlern führen kann, enthält Oberon eine entsprechende Funktion nicht. Hier kann man sich mit der Funktion *ADR* aus dem Modul *SYSTEM* helfen. In vielen Fällen ist es jedoch nicht wünschenswert, *SYSTEM* zu importieren, und damit anzuzeigen, daß es sich um ein systemnahes Modul handelt.

Viele Parameter von Library-Routinen sind daher in den Interface-Modulen nicht als Adressen, sondern als die Strukturen der übergebenen Werte selbst definiert. Bei solchen strukturierten Registerparametern für Library-Prozeduren erzeugt der Compiler automatisch Code, der die Adresse des übergebenen Wertes übergibt. In dem Beispielprogramm *Weekday* oben wurde dies bereits ausgenutzt: Es wird hier einfach die Variable *date* an *DateStamp* übergeben, in 'C' müßte hier die Adresse der Variablen übergeben werden.

Ähnlich verhält es sich mit der Routine *CopyMem* aus der Exec-Library. Sie ist im Interface-Modul *Exec* folgendermaßen definiert:

```
PROCEDURE CopyMem*{exec,-624}(
                    source{8} : ARRAY OF BYTE;
                    dest{9}   : ARRAY OF BYTE;
                    size{0}   : LONGINT);
```

Die Parameter geben hier also direkt die Variablen an, die kopiert werden sollen, und nicht deren Adressen. Sollen die Speicherbereiche, auf

die die zwei Zeigervariablen *p* und *q* zeigen, kopiert werden, so muß der Aufruf *CopyMem(p^,q^,len)* heißen. Benötigt man dennoch einmal einen Aufruf von *CopyMem*, bei dem Adressen als Parameter übergeben werden, kann die Prozedur *CopyMemAPTR* aus dem Interface-Modul *Exec* verwendet werden. Dies ist dieselbe Prozedur, ihre Parameter sind jedoch als *APTR* definiert.

Eine Sonderrolle nehmen Prozeduren ein, die als Parameter die Adressen von Zeichenketten erwarten. In den Interface-Modulen sind die Parameter hier als *ARRAY OF CHAR* definiert, so daß die Zeichenketten direkt übergeben werden können. Vorsicht ist hier jedoch angebracht, da alle die Zeichenketten, die dem Betriebssystem übergeben werden, mit dem Zeichen *0X* abgeschlossen sein müssen. Daher darf eine Zeichenkette, die in einer Variablen des Typs *ARRAY 80 OF CHAR* gespeichert wird und an eine Betriebssystemroutine übergeben wird, maximal 79 Zeichen enthalten, das letzte Zeichen wird für das Abschlußzeichen verwendet.

Die Prozedur *FindTask* aus *Exec* erhält eine Zeichenkette als Parameter. Sie ist in *Exec* wie folgt definiert:

```
PROCEDURE FindTask*{exec,-294}(
                name{9} : ARRAY OF CHAR): TaskPtr;
```

So kann ein Zeiger auf die Task-Struktur des Collector-Prozesses mit der Anweisung

```
collector := Exec.FindTask("recycling memory");
```

bestimmt werden (da dies ein Task der Garbage-Collector-Library ist, der diesem Programm nicht gehört und über dessen Lebensdauer wir nichts wissen, dürfen wir mit dem erhaltenen Zeiger jedoch nicht arbeiten).

FindTask kann jedoch in 'C' auch einfach der Wert *NULL* übergeben werden, um anzuzeigen, daß man einen Zeiger auf die Task-Struktur

des eigenen Programms benötigt. Daher erlaubt Amiga Oberon die Übergabe von *NIL* an einen Registerparameter des Typs *ARRAY OF CHAR* beim Aufruf einer Library-Prozedur. Einen Zeiger auf den eigenen Task erhält man also mit der Anweisung:

```
Me := Exec.FindTask(NIL);
```

Tag-Listen

Ab AmigaOS 2.0 gibt es Library-Routinen die sogenannte Tag-Listen als Parameter erwarten. Dies sind im Prinzip Adressen von beliebig langen Feldern, die jeweils Paare von zwei 32-Bit Werten enthalten. Damit solche Prozeduren leicht in Oberon-Programmen aufgerufen werden können, kennt Amiga Oberon Listenparameter. Diese wurden in Kapitel 15 beschrieben.

Zusätzlich enthalten die Interface-Module noch weitere Versionen der Prozeduren mit Tag-Listen, die Felder von *Utility.TagItem* als Parameter erwarten. So enthält *ASL* die Prozedur *AllocAslRequest* doppelt, einmal unter dem Namen *AllocAslRequest* und mit einem Feld als Parameter, und mit dem Namen *AllocAslRequestTags* mit einer Tagliste. Der folgende Aufruf mit einer Tagliste

```
fr := ASL.AllocAslRequestTags(ASL.fileRequest,
                              ASL.leftEdge,20,
                              ASL.topEdge, 20,
                              ASL.width,   280,
                              ASL.height,  160,
                              Utility.done);
```

ist dabei völlig gleichwertig mit dem Aufruf von *AllocAslRequest*, bei dem ein Feld von *Utility.TagItem* übergeben wird:

```
fr := ASL.AllocAslRequest(ASL.fileRequest,
                Utility.Tags5(ASL.leftEdge,20,
                              ASL.topEdge, 20,
                              ASL.width,   280,
                              ASL.height,  160,
                              Utility.done,0));
```

In den meisten Fällen ist es letztendlich Geschmacksache, welche der beiden Aufrufmethoden verwendet wird. Manchmal ist der Aufruf mit einem Feld sinnvoller, da man hier eine Variable anlegen kann, die alle TagItems enthält und schon vor dem Prozeduraufruf ausgefüllt werden kann.

BPOINTER in Taglisten

Besondere Vorsicht ist geboten, wenn ein *BPOINTER*, beispielsweise ein *FileHandlePtr* aus *Dos*, in einer Tagliste übergeben werden soll. Da die Taglisten in den Interface-Modulen mit dem Typ *SYSTEM.ADDRESS* definiert sind, wird der *BPOINTER* in einen gewöhnlichen Zeiger umgewandelt, bevor er an die aufgerufene Prozedur übergeben wird. Gewöhnlich erwarten die Routinen des Betriebssystems hier jedoch einen *BPOINTER*. Um die Umwandlung zu verhindern, muß der Typ mit *SYSTEM.VAL* umgewandelt werden. Soll beispielsweise die Variable *file* übergeben werden, ist der Ausdruck *SYSTEM.VAL(SYSTEM.ADDRESS,file)* nötig.

Vorzeichenlose Integerzahlen

Oberon kennt für ganze Zahlen lediglich die Typen *SHORTINT*, *INTEGER* und *LONGINT*, die jeweils vorzeichenbehaftet sind. Die Routinen des AmigaOS kennen jedoch auch die vorzeichenlosen Zahlentypen *UBYTE*, *USHORT* und *ULONG*. Diese Zahlen sind immer positiv und haben für die positiven Zahlen einen größeren Wertebereich als die Integer-Zahlen von Oberon.

In vielen Fällen werden die vorzeichenlosen Zahlen als Mengentypen verwendet. Sie sind in den Interface-Modulen dann als *SHORTSET*, *SET* oder *LONGSET* definiert. So kann der 'C'-Ausdruck *WFLG_ACTIVATE | WFLG_BORDERLESS* in Oberon einfach als *LONGSET{I.activate,I.borderless}* geschrieben werden (hier wurde davon ausgegangen, daß die Importliste den Eintrag *I := Intuition* enthält). Manche Definitionen der Mengenelemente in den Interface-Modulen enthalten auch fertige Mengen, in denen mehrere Elemente gesetzt sind. Diese müssen mit dem Mengenvereinigungsoperator '+' zusammengefügt werden, möchte man also bei der Menge von oben zusätzlich *otherRefresh* setzen, so muß der Ausdruck *LONGSET{I.activate,I.borderless} + I.otherRefresh* lauten.

Größere Probleme bereiten die vorzeichenlosen Typen, wenn sie wirklich als Variablen für Zahlen verwendet werden. Haben sie die Größe ein Byte, so sind sie in den Interface-Modulen als *SYSTEM.BYTE* definiert. Dann können sie leicht mit *ORD* in eine *INTEGER*-Zahl zwischen *0* und *255* umgewandelt und mit *CHR* wieder zurückgewandelt werden.

Vorzeichenlose Werte, die zwei oder vier Bytes belegen, sind auch in den Interface-Modulen als *INTEGER* bzw. *LONGINT* definiert. Da der Wertebereich der Zahlen meist nicht voll ausgenutzt wird, kann in vielen Fällen einfach mit diesen vorzeichenbehafteten Zahlen gearbeitet werden. Bei solchen *INTEGER*-Werten wird jedoch manchmal der volle Wertebereich der vorzeichenlosen Zahl benötigt. Dies ist beispielsweise bei den Variablen *horizPot*, *vertPot*, *horizBody* und *vertBody* der *PropInfo*-Struktur in *Intuition* der Fall. Um mit diesen Werten in Oberon zu arbeiten, bietet das Interface-Modul zu *Intuition* zwei Prozeduren an:

PROCEDURE UIntToLong(i: INTEGER): LONGINT;

Diese Prozedur erhält einen *INTEGER*-Parameter *i* der eine vorzeichenlose Zahl darstellt. Das Ergebnis ist der *LONGINT*-Wert dieser Zahl. Der Wert liegt zwischen *0* und *0FFFFX*.

PROCEDURE LongToUInt(l: LONGINT): INTEGER;

Der *LONGINT*-Parameter *l* muß zwischen *0* und *0FFFFX* liegen. Er wird in einen *INTEGER*-Wert umgewandelt, der eine vorzeichenlose Zahl darstellt.

Mit den vorzeichenlosen Zahlen selbst darf nicht gerechnet werden. Da der Wertebereich der *LONGINT*-Zahlen jedoch groß genug ist, kann stattdessen mit dem von *UIntToLong* erhaltenen Wert gearbeitet werden. Entsprechend kann der Wert dann mit *LongToUInt* wieder zurückgewandelt werden.

Vorzeichenlose 32-Bit Zahlen, bei denen der Wertebereich voll ausgeschöpft wird, kommen im AmigaOS praktisch nicht vor, so daß Amiga Oberon und die Interface-Module hier keine spezielle Sonderbehandlung anbieten.

Arbeiten mit Devices

Neben den Libraries sind die Devices wichtige Komponenten des AmigaOS. Die gesamte Ein- und Ausgabe wird über Devices abgewickelt. Ein Device wird mit der Exec-Routine *OpenDevice* geöffnet. Dazu muß zunächst ein Port zur Kommunikation mit dem Device angelegt und ein IORequest-Block ausgefüllt werden. Zusätzlich sind, abhängig von dem Device, mit dem gearbeitet wird, verschiedene Initialisierungen nötig. So kann das Audio-Device mit folgender Prozedur geöffnet werden. Dabei wird gleich ein Audio-Kanal angefordert:

```
TYPE
  Map = ARRAY 4 OF SHORTSET;
VAR
  Port: Exec.MsgPortPtr;
  IO:  Audio.IOAudioPtr;
  Open: BOOLEAN;
  map: UNTRACED POINTER TO Map;
```

```
PROCEDURE OpenAudio(): BOOLEAN;
BEGIN
  Port := ExecSupport.CreatePort("",0);
  IF Port=NIL THEN RETURN FALSE END;
  NEW(IO); NEW(map);
  map^ := Map(SHORTSET{0},SHORTSET{3},
                  SHORTSET{1},SHORTSET{2});
  IO.request.message.node.pri   := -40;
  IO.request.message.replyPort := Port;
  IO.data := map; IO.length := 4;
  Open := (Exec.OpenDevice("audio.device",0,IO,
  LONGSET{})=0) & (IO.request.error = 0);
  RETURN Open;
END OpenAudio;
```

Geschlossen wird ein Device entsprechend mit *CloseDevice* aus *Exec*. Danach kann auch der Port wieder freigegeben werden. Um das oben geöffnete Audio-Device zu schließen, kann beispielsweise folgende Routine verwendet werden:

```
PROCEDURE CloseAudio();
BEGIN
  IF Open THEN
    Exec.CloseDevice(IO);
    Open := FALSE;
  END;
  IF Port#NIL THEN
    ExecSupport.DeletePort(Port)
  END;
END CloseAudio;
```

Um mit dem Device zu arbeiten, müssen ihm mit den Prozeduren *DoIO*, *SendIO* aus *Exec* oder *BeginIO* aus *ExecSupport* Commandos in Form von IORequest-Blöcke geschickt zu werden. Folgende Routine spielt mit dem Audio-Device, das mit der obigen Routine geöffnete wurde, einen kurzen Ton:

```
TYPE
  Table = ARRAY 2 OF SHORTINT;
VAR
  table : UNTRACED POINTER TO Table;

PROCEDURE Sound(period: INTEGER);
BEGIN
  IF table=NIL THEN
    INCL(OberonLib.MemReqs,Exec.chip);
    NEW(table);
    EXCL(OberonLib.MemReqs,Exec.chip);
    table^ := Table(127,-128);
  END;
  IO.request.command := Exec.write;
  IO.request.flags   := SHORTSET{Audio.pervol};
  IO.data            := table;
  IO.length          := 2;
  IO.volume          := 64;
  IO.period          := period;
  IO.cycles          := SHORT(500000 DIV period);
  ExecSupport.BeginIO(IO);
  IF Exec.WaitIO(IO)=0 THEN END;
END Sound;
```

Hier wird die Kombination der Aufrufe von *BeginIO* und *WaitIO* verwendet. Bei anderen Devices wird gewöhnlich stattdessen *DoIO* verwendet. Mit den Prozeduren von oben kann nun z.B. mit der folgenden Anweisung gearbeitet werden:

```
IF OpenAudio() THEN
  Sound(3000);
  Sound(3367);
  Sound(3780);
END;
CloseAudio;
```

Manche Devices bietet Routinen ähnlich denen der Libraries an. Diese Prozeduren werden in den Interface-Modulen der Devices definiert,

jedoch öffnen die Module die Devices nicht selbst. Stattdessen müssen die Benutzer eines Devices die Basisadresse des Devices in eine Variable des Interface-Moduls schreiben. Danach können die Prozeduren des Devices verwendet werden.

Das Console-Device ist beispielsweise ein solches Device. Um die Routine *RawKeyConvert* dieses Devices aufzurufen, sind folgende Anweisungen nötig:

```
IF e.OpenDevice(Console.consoleName,
                -1,ioreq,LONGSET{})=0 THEN
  Console.base := ioreq.device;
  n := Console.RawKeyConvert(ev,s,l,NIL);
END;
```

Vorsicht ist hier nötig, da die angegebene Aufrufmöglichkeit nicht reentrant ist. Es dürfen niemals zwei Prozesse gleichzeitig auf diese Weise mit demselben Device arbeiten.

Hook-Funktionen

Ab AmigaOS 2.0 gibt es für Programme die Möglichkeit, Prozeduren in Form von Hook-Funktionen an Routinen des Betriebssystems zu übergeben. Diese Prozeduren werden dann von den Betriebssystemroutinen aufgerufen. Eine solche Prozedur wird mit einem Objekt des Typs *Utility.Hook* beschrieben.

Oberon-Module, die solche Hook-Funktionen verwenden, und mit dem kleinen Datenmodell arbeiten, müssen mit besonderer Vorsicht geschrieben werden. Die Hook-Funktion muß das Register A5 auf die Basisadresse der globalen Variablen setzen, bevor sie auf globale Variablen zugreift. Da die Routine *StackChk* aus *OberonLib*, die die Stapelkontrolle erledigt, auf eine globale Variable zugreift, ist hier besondere Vorsicht nötig: Die erste, als Hook-Funktion aufgerufene Prozedur muß daher ohne Stapelkontrolle übersetzt werden.

Da eine Hook-Funktion eventuell von einem anderen Prozeß als dem des Oberon-Programms aufgerufen wird, darf innerhalb dieser nicht mit verfolgten Zeigern gearbeitet werden, der Garbage-Collector kennt den anderen Prozeß nicht als einen der Mutatoren.

Damit mit Hook-Funktionen etwas leichter gearbeitet werden kann, bietet das Interface-Module *Utility* die Prozedur *InitHook* zur Initialisierung einer Hook-Struktur an. Die hier angegebene Prozedur hat gewöhnliche Parameter, keine Registerparameter. A5 wird automatisch in *hookdata* gespeichert und korrekt gesetzt, so daß innerhalb der Prozedur, bis auf den Verzicht auf das Arbeiten mit verfolgten Referenzen, nichts besonderes beachtet werden muß.

Das folgende Programm gibt die Namen und Größen aller Dateien im aktuellen Verzeichnis aus. Das Verzeichnis wird dabei mit der Dos-Funktion *ExAll* eingelesen. Über eine Hook-Funktion wird sichergestellt, daß nur Dateien, und keine Verzeichnisse ausgegeben werden:

```
MODULE Files;

IMPORT Dos, Utility, Exec, OberonLib;

VAR
  buffer: Dos.ExAllDataPtr;
  control: Dos.ExAllControlPtr;
  more: BOOLEAN;
  data: Dos.ExAllDataPtr;
  ln: BOOLEAN;
  cd: Dos.FileLockPtr;

CONST
  bufsize = 2048;

  PROCEDURE MatchFunk(hook: Utility.HookPtr;
                      type: Exec.APTR;
                      match: Exec.APTR): LONGINT;
  VAR m: Dos.ExAllDataPtr;
```

```
  BEGIN
    m := match;
    IF m.type = Dos.file THEN RETURN -1
                          ELSE RETURN  0 END;
  END MatchFunk;

BEGIN
  IF Dos.dos.lib.version>=37 THEN
    IF ~ OberonLib.wbStarted THEN
      control := Dos.AllocDosObjectTags(
                       Dos.exAllControl);
      IF control=NIL THEN
        Dos.PrintF("AllocDosObject failed!\n");
      ELSE
        control.lastKey := 0;
        NEW(control.matchFunc);
        Utility.InitHook(control.matchFunc,
                         MatchFunk);
        OberonLib.New(buffer,bufsize);
        ln := FALSE;
        cd := Dos.Lock("",Dos.sharedLock);
        IF cd=NIL THEN
          Dos.PrintF("Lock failed!\n");
        ELSE
          REPEAT
            more := Dos.ExAll(cd,buffer^,bufsize,
                              Dos.size,control);
            IF ~more &
               (Dos.IoErr()#Dos.noMoreEntries) THEN
              Dos.PrintF("Fehler!\n");
            ELSE
              IF control.entries>0 THEN
                data := buffer;
                REPEAT
                  Dos.PrintF("%-16s %8ld    ",
                             data.name,data.size);
                  IF ln THEN Dos.PrintF("\n") END;
                  ln := ~ln;
                  data := data.next;
                UNTIL data=NIL;
```

```
            END;
          END;
        UNTIL ~more;
        IF ln THEN Dos.PrintF("\n") END;
        Dos.UnLock(cd);
      END;
      Dos.FreeDosObject(Dos.exAllControl,
                        control);
    END;
  END;
END;
END Files.
```

Dieses Programm funktioniert nur dann, wenn es von einer Shell aus gestartet wurde und wenn die Dos-Library mindestens die Version 37 hat. Zum vollständigen Verständnis dieses Programms ist die Kenntnis der Routinen der Dos-Library unbedingt erforderlich. Eine Dokumentation der Dos-Library wird beispielsweise in [AmigaDos 91] gegeben.

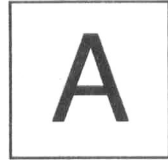
MatchFunc ist hier die Prozedur, die als Hook-Funktion verwendet wird. Sie bekommt mit dem Parameter *match* einen Zeiger auf den aktuellen Eintrag des Verzeichnisses. Über den Rückgabewert, *-1* oder *0*, wird entschieden, ob dieser Eintrag in die von *ExAll* erzeugte Liste aufgenommen werden soll. Um diese Hook-Funktion zu aktivieren, wird zunächst mit *NEW(control.matchFunc)* ein Hook-Objekt erzeugt. Dieses wird mit *Utility.InitHook(control.matchFunc,MatchFunc)* initialisiert.

Da *MatchFunc* zuweisungskompatibel zum Prozedurparameter von *InitHook* sein muß, kann der Parameter *match* nicht als *Dos.ExAllDataPtr* definiert werden, sondern muß *Exec.APTR* sein. Durch die Zuweisung dieses Parameters an eine lokale Variable des Typs *Dos.ExAllDataPtr* kann dennoch leicht auf die Daten des aktuellen Eintrags zugegriffen werden.

Nachdem die Hook-Funktion installiert ist, wird sie beim Aufruf von *Dos.ExAll* von Dos aus aufgerufen.

Einen kleinen Trick enthält das Programm noch, der hier einen Import von *SYSTEM* vermeidet: Der von *ExAll* erwartete Puffer wird mit der Prozedur *New* aus *OberonLib* mit der Größe 2048 Bytes angefordert. Damit die Variable, die den Zeiger auf den Puffer enthält, gleich als Zeiger auf das erste Element der Liste der Verzeichniseinträge verwendet werden kann, wird sie als *Dos.ExAllDataPtr* definiert, obwohl sie in Wirklichkeit auf ein größeres Objekt zeigt.

Anhang A: Fehlermeldungen



Amiga Oberon kennt die im folgenden aufgelisteten Fehlermeldungen. Sie sind in der Datei 'Oberon:Fehler-Meldungen' als gewöhnlicher Text gespeichert. Dieser Text wird von den Programmen OEd (Kapitel 3 und 4), OErr (Kapitel 7) und der 'oberonsupport.library' zum Anzeigen der Meldungen, die der Compiler in der Datei mit der Endung '.modE' gespeichert hat, benutzt.

Die Fehlermeldungen

Nummer	Fehlermeldung
0:	Unerwartetes Zeichen
1:	---
2:	Bezeichner zu lang (Parserfehler)
3:	Zahl zu lang (Parserfehler)
4:	Fehler in Ganzzahl
5:	Fehler in Realzahl
6:	Zahl zu groß
7:	"*)" erwartet
8:	Steuerzeichen in konstantem String
9:	MODULE erwartet
10:	Modulname erwartet
11:	Strichpunkt erwartet
12:	Bezeichner an Modulkopf und Modulende verschieden
13:	Punkt am Ende der Compilationseinheit erwartet
14:	Modulname am Modulende erwartet
15:	END an Modulende erwartet
16:	"]" erwartet
17:	"=" in Konstanten- oder Typdeklaration erwartet
18:	---
19:	Inkompatible Operanden
20:	Bezeichner vor IN muß zwischen 0 und MAX(Set) liegen
21:	Bezeichner nach IN muß SET sein
22:	Bezeichner vor IN muß INTEGER sein
23:	---
24:	---
25:	Bezeichner nicht definiert
26:	Konstanter Ausdruck erwartet
27:	")" erwartet
28:	"~" kann nur auf boolsche Operanden angewendet werden

29: ---
30: ---
31: ---
32: Setelement zu groß oder zu klein
33: Zweites Element muß größer als erstes sein
34: "}" erwartet
35: Bezeichner bereits definiert
36: Bezeichner erwartet
37: ---
38: Typ erwartet
39: Typ muß INTEGER sein
40: ---
41: ---
42: OF erwartet
43: ":" erwartet
44: END erwartet
45: ---
46: TO erwartet
47: Typ, auf den Pointer, zeigt ist nicht definiert
48: "(" erwartet
49: Funktionsergebnis muß unstrukturiert sein
50: Bezeichner am Prozedurende erwartet
51: Bezeichner am Prozedurende und Prozeduranfang verschieden
52: Bezeichner für Prozedurname schon definiert
53: ---
54: ---
55: ---
56: Typ muß RECORD sein
57: Recordelementbezeichner erwartet
58: Recordelement existiert nicht oder ist nicht sichtbar
59: Typ muß Pointer sein
60: Typ muß Array sein
61: Feldindextyp muß INTEGER sein
62: ---
63: Vorzeichen nur bei Zahlen
64: Adressierungsart nicht erlaubt (Compilerfehler)
65: OR und AND nur für BOOLEAN definiert
66: ---
67: Faktor erwartet
68: Prozeduraufruf mit zu wenig Parametern
69: Parameter hat falschen Typ
70: Zu viele Parameter
71: ---
72: Funktionsaufruf einer Prozedur
73: "," erwartet
74: ---
75: ---
76: Zugewiesener Wert hat falschen Typ

77: Typ sollte Prozedur sein
78: Prozeduraufruf einer Funktion
79: Typ hinter IF, WHILE und UNTIL muß BOOLEAN sein
80: THEN erwartet
81: Typ hinter CASE muß INTEGER oder CHAR sein
82: ---
83: ---
84: DO hinter WHILE / WITH erwartet
85: UNTIL in REPEAT-Anweisung erwartet
86: ":=" erwartet
87: ---
88: RETURN nur innerhalb von Prozeduren erlaubt
89: Typ hinter RETURN muß gleich dem Prozedur-Ergebnistyp sein
90: "." erwartet
91: Feldindex zu groß oder zu klein
92: Noch nicht implementiert
93: Qualident hat falschen Typ
94: 32K-Bereich überschritten (Modulcode)
95: Nicht genügend Register verfügbar
96: Konnte Symboldatei nicht laden
97: ---
98: EXIT nur innerhalb von LOOP Anweisungen erlaubt
99: Zuweisung nur an Variable möglich
100: In ursprünglicher Definition waren andere Parameter
101: In ursprünglicher Definition waren mehr/weniger Parameter
102: Ursprüngliche Definition hatte anderes Funktionsergebnis
103: Adresse nicht ladbar
104: ---
105: Parameter von ASH muß INTEGER sein
106: Parameter von CAP muß CHAR sein
107: Parameter von CHR muß INTEGER oder SYSTEM.BYTE sein
108: Parameter von DEC und INC muß INTEGER sein
109: DEC/INC kann nur auf Variablen angewendet werden
110: Bezeichner sollte Typ sein
111: Librarybase muß Variable sein
112: Der 68000'er hat nur 16 Register mit den Nummern 0 bis 15
113: Stack- und Registerparameter dürfen nicht gemischt werden
114: ---
115: ---
116: Typumwandlung nur bei Typen mit gleicher Größe
117: Register nicht freigegeben (Compilerfehler)
118: Implementation einer Forward-Prozedur fehlt
119: Doppeltes CASE-Element
120: Elemente in CASE müssen Konstanten sein
121: Bezeichner sollte Prozedur sein
122: Parameter von NEW, ALLOCATE, DISPOSE und INIT muß Pointer sein
123: Adresse nicht ladbar (Implementationsbeschränkung)
124: ---

125: ---
126: Erster Parameter von LEN muß Feld sein
127: Zweiter Parameter von LEN zu groß
128: ---
129: Settypbezeichner erwartet
130: ---
131: Parameter von ABS muß numerisch sein
132: Bereichsfehler
133: Parameter von ENTIER muß REAL sein
134: Konstante nicht ladbar
135: Dreistellige Octalahl in String erwartet
136: ---
137: Index bei offenen Feldern muß INTEGER sein
138: Versionskonflikt
139: zweistellige Hexzahl in String erwartet
140: TypeGuard-Konflikt (Compilerbeschr.), ändere den Modulnamen
141: Zu viele RECORDs in diesem Modul (Compilerbeschränkung)
142: Recordelementbezeichner existiert schon
143: Bezeichner hinter IS sollte Typ sein
144: ---
145: Bei "v IS T", "v(T)" und "WITH v:T DO" muß v Basistyp von T sein
146: Typeguard muß Recordtypbezeichner sein
147: WITH v:T DO: v muß Variable sein
148: WITH v:T DO: T muß Erweiterung von v sein
149: Division durch Null
150: 1. Parameter von LSH und ROT muß SET sein
151: 2. Parameter von ASH, LSH und ROT muß INTEGER sein
152: Benötigter Stapelbereich zu groß (>32K)
153: SIZE darf nicht auf offene Felder angewendet werden
154: Prozedurkörper in implementationslosen Modulen verboten
155: Prozedur war in Forward-Declaration (nicht) exportiert
156: Es dürfen keine rekursiven Typen definiert werden
157: Parameter von ODD() muß Integer sein
158: ---
159: Lokale Prozeduren können nicht exportiert werden
160: Lokale Prozeduren können keinen Variablen zugewiesen werden
161: ---
162: Variablenbereich > 32K bei SmallData
163: ---
164: Illegale Konstante
165: Nicht genügend Speicher vorhanden
166: Parameter muß LONGREAL sein
167: ---
168: Recordtyp zu groß
169: Listenparameter ist hier nicht erlaubt
170: Listenparameter muß letzter Parameter sein
171: ---
172: Typ muß STRUCT sein

173: ---
174: Kann mit strukturierter Registervariablen hier nicht arbeiten
175: Neue Symboldatei muß erzeugt werden
176: Konstanten mit Referenzen sind in impl-losen Modulen verboten
177: Registerparameter dürfen keine offenen Felder sein.
178: Receiver muß Zeiger auf RECORD oder RECORD VAR-Parameter sein
179: Registerparameter bei typgebundenen Prozeduren nicht erlaubt.
180: Prozedur wurde nicht neu definiert
181: Typ und typgebundene Prozeduren müssen im selben Modul definiert werden
182: Variable ist nur zum Lesen exportiert
183: Bezeichner nach FOR muß Integer-Variable sein
184: Typ hinter BY muß zuweisungskompatibel zur Laufvariablen sein
185: Zeiger auf offene Felder müssen direkt definiert werden
186: Typ darf kein offenes Feld sein
187: POINTER erwartet
188: UNTRACED POINTER und BPOINTER müssen auf nicht verfolgte Objekte zeigen
189: Ziel von FOR muß gleichen Typ wie Laufvariable haben
190: Typ ist zu groß
191: Es dürfen maximal 8191 Methoden pro RECORD definiert werden
192: Zyklischer Import ist nicht erlaubt
193: Registervariablen der umschließenden Prozedur dürfen nicht benutzt werden
194: $JOIN-Datei nicht gefunden

Format der Fehlerdatei

Die vom Compiler erzeugte Fehlerdatei, die mit der Endung '.modE' gespeichert wird, ist sehr einfach aufgebaut. Die Daten in der Datei können als ein Feld von Structs des folgenden Typs angesehen werden:

```
TYPE
  FehlerDatei = ARRAY Fehleranzahl OF STRUCT
    num, line, column : INTEGER;
  END;
```

Dabei ist *num* die Nummer des Fehlers in der obigen Liste und gleichzeitig die Zeile der Fehlermeldung in der Datei 'Oberon:Fehler-Meldungen' (dabei hat die oberste Zeile die Nummer null).

line und *column* geben die Position des Fehlers im Quelltext an, wobei die oberste Zeile und die erste Spalte jeweils die Nummer eins haben. Ist *column* < *1* so ist der Fehler am Ende der vorigen Zeile.

Anhang B: Syntax von Amiga Oberon

B

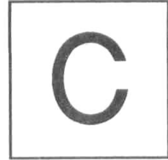
Durch die Erweiterung des Sprachumfangs von Amiga Oberon (Kapitel 14 und 15) hat sich auch die Syntax der Sprache verändert. Im folgenden wird eine Definition der von Amiga Oberon verarbeiteten Sprache in EBNF-Notation (erweiterte Backus-Naur-Form, siehe [Reiser 92]) gegeben:

```
CompilationUnit   = { Module } .
Module            = MODULE Ident ";" [ImportList]
                    DeclSequence
                    [ BEGIN StatementSequence ]
                    [ CLOSE StatementSequence ]
                    END Ident "." .
ImportList        = IMPORT Import { "," Import } ";" .
Import            = IdentDef [ (":=" | ":") Ident ] .
IdentDef          = Ident [ "*" | "-" ] .
QualIdent         = [ Ident "." ] Ident .
Ident             = Letter { Letter | Digit } .
Letter            = "a" | .. | "z" | "A" | .. | "Z" .
Digit             = "0" | .. | "9" .
HexDigit          = Digit | "A" | "B" | "C" | "D" | "E" | "F" .
StringConstant    = '"' {Character} '"' { '"' {Character} '"' }
                    | "'" {Character} "'" { "'"{Character}"'" } .
CharConstant      = '"' Character '"' | "'" Character "'"
                    | Digit { HexDigit } "X" .
Number            = Integer | Real .
Integer           = Digit {Digit}
                    | Digit {HexDigit} ( "H" | "U" ) .
Real              = ( Digit {Digit} "." {Digit}
                    | "." Digit {Digit} )
                    [ ("D" | "E") ["+" | "-"] {Digit} ] .
DeclSequence      = { CONST { ConstantDecl ";" }
                    | TYPE { TypeDeclaration ";" }
                    | VAR { VariableDecl ";" }
                    | { ProcDeclaration ";" } } .
ProcDeclaration   = ProcedureDecl
                    | ForwardDecl
                    | ExternalDecl .
ConstantDecl      = IdentDef "=" ConstExpression .
ConstExpression   = Expression .
TypeDeclaration   = IdentDef "=" Type .
VariableDecl      = VariableIdent ":" Type .
VariableIdent     = IdentDef [ "[" ( StringConstant |
                    ConstExpression ) "]" ] .
```

```
ProcedureDecl       = ProcedureHeading ";"
                      ProcedureBody Ident .
ProcedureHeading    = PROCEDURE ["*"] [Receiver]
                      IdentDef [FormalParameters] .
ProcedureBody       = DeclSequence
                      [ BEGIN StatementSequence ] END .
Receiver            = "(" [VAR] Ident ":" Ident ")" .
ForwardDecl         = PROCEDURE "^" [Receiver] IdentDef
                      [FormalParameters] .
ExternalDecl        = PROCEDURE ["*"] [Receiver] IdentDef
                      "{" ( StringConstant | Ident ","
                      ConstExpression ) "}" [FormalParameters] .
FormalParameters    = "(" [ FPSection { ";" FPSection } ] ")"
                      [ ":" QualIdent ] .
FPSection           = [VAR] Parameter { "," Parameter }
                      ":" Type .
Parameter           = Ident [ "{" ConstExpression "}" [".."] ] .
Type                = QualIdent
                      | ArrayType
                      | RecordType
                      | StructType
                      | PointerType
                      | ProcedureType .
ArrayType           = ARRAY [ ConstExpression
                      { "," ConstExpression } ] OF Type .
RecordType          = RECORD [ "(" Qualident ")" ]
                      FieldListSeq END .
StructType          = STRUCT [ "(" IdentDef ":" Qualident ")" ]
                      FieldListSeq END .
FieldListSeq        = FieldList { ";" FieldList } .
FieldList           = [ IdentDef { "," IdentDef } ":" Type ] .
PointerType         = ( [UNTRACED] POINTER | BPOINTER ) TO Type .
ProcedureType       = PROCEDURE [FormalParameters] .
Expression          = SimpleExpression
                      [Relation SimpleExpression] .
Relation            = "=" | "#" | "<" | "<=" |
                      ">" | ">=" | IN | IS .
SimpleExpression    = [ "+" | "-" ] Term { AddOperator Term } .
AddOperator         = "+" | "-" | OR .
Term                = Factor { MulOperator Factor } .
MulOperator         =  "*" | "/" | DIV | MOD | "&" | AND .
Factor              = Number
                      | CharConstant
                      | StringConstant
                      | Set
                      | Designator
                      | "(" Expression ")"
                      | ( "~" | NOT ) Factor .
Set                 = [QualIdent] "{" [Elements] "}" .
Elements            = Element { "," Element } .
```

```
Element           = Expression [ ".." Expression ] .
Designator        = QualIdent { "." Ident | "[" ExpList "]"
                    | "(" [ExpList] ")" | "^" } .
ExpList           = Expression { "," Expression } .
StatementSequence= Statement { ";" Statement } .
Statement         = [ Assignment
                    | ProcedureCall
                    | IfStatement
                    | CaseStatement
                    | WhileStatement
                    | RepeatStatement
                    | LoopStatement
                    | WithStatement
                    | ForStatement
                    | EXIT
                    | RETURN [Expression] ] .
Assignment        = Designator ":=" Expression .
ProcedureCall     = Designator .
IfStatement       = IF Expression THEN StatementSequence
                    { ELSIF Expression THEN
                    StatementSequence }
                    [ ELSE StatementSequence ] END .
CaseStatement     = CASE Expression OF Case { "|" Case }
                    [ ELSE StatementSequence ] END .
Case              = [ Elements ":" StatementSequence ] .
WhileStatement    = WHILE Expression DO StatementSequence END .
RepeatStatement   = REPEAT StatementSequence UNTIL Expression .
LoopStatement     = LOOP StatementSequence END .
WithStatement     = WITH Guard DO StatementSequence
                    { "|" Guard DO StatementSequence }
                    [ ELSE StatementSquence ] END .
Guard             = QualIdent ":" QualIdent .
ForStatement      = FOR Ident ":=" Expression TO Expression
                    [ BY ConstExpression ]
                    DO StatementSequence END .
```

Anhang C: Tabellen



Dieser Anhang enthält die wichtigsten Tabellen aus verschiedenen Kapiteln des Handbuchs.

Piktogramm-Merkmale von OEd

Merkmal:	Bedeutung:
ICONS	Piktogramme erzeugen
TABULATOR	Tabulatorweite = n
LEFTEDGE	Horizontale Position der Textfenster
TOPEDGE	Vertikale Position der Textfenster
WIDTH	Breite der Textfenster
HEIGHT	Höhe der Textfenster
DEPTH	Tiefe des OEd-Bildschirms
SCREEN	OEd soll eigenen Bildschirm öffnen
INTERLACE	Bildschirm im Interlace-Modus öffnen
MAXUNDO	Größe des 'undo'-Puffers
LAYOUT	Setzt den Schalter 'layout'
AUTOUPPERCASE	Setzt den Schalter 'autouc'

Shell-Optionen von OEd

Option:	Bedeutung
-i	Piktogramme erzeugen
-t#	Tabulatorbreite auf # setzen.
-x#,-y#	x- und y-Position der OEd-Fenster
-w#,-h#	Breite und Höhe der OEd-Fenster
-d#	Tiefe des OEd-Bildschirms
-s	OEd soll eigenen Bildschrim öffnen
-l	Bildschirm im Interlace-Modus öffnen
-u#	Maximale Anzahl der Textänderungen, die sich OEd merken soll und die mit dem Kommando 'undo' rückgängig gemacht werden können. Voreingestellt ist 50.
-o	deaktiviert den Schalter 'layout'
-a	aktiviert den Schalter 'autouc'

Piktogramm-Merkmale des Compilers

Option:	Bedeutung:
STACKCHK	Stackkontrolle
OVFLCHK	Überlaufskontrolle
RANGECHK	Bereichskontrolle
CASECHK	Case-Index-Kontrolle
RETURNCHK	Return-Kontrolle
NILCHK	NIL-Zeiger-Kontrolle
ODDCHK	Ungerade-Zeiger-Kontrolle
TYPECHK	Typkontrolle
CLEARVARS	Lokale Variablen löschen
SMALLCODE	Kleines Codemodell
SMALLDATA	Kleines Datenmodell
MC68010	Code für MC68010 erzeugen
MC68020	Code für MC68020 erzeugen
MC68030	Code für MC68030 erzeugen
MC68881	Code für FPU MC68881/2 erzeugen
GARBAGECOLLECTOR	Garbage-Collector benutzen
EXTENSIONS	Spracherweiterungen benutzen
DEBUG	Code für Debugger ODebug
NEWSYMBOLS	Neue Symboldatei erzeugen
ICONS	Piktogramme erzeugen

Shell-Optionen des Compilers

Option:	Bedeutung:	Vorgabe:
-s	Stackkontrolle	TRUE
-v	Überlaufskontrolle	TRUE
-b	Bereichskontrolle	TRUE
-c	Case-Index-Kontrolle	TRUE
-r	Return-Kontrolle	TRUE
-n	NIL-Zeiger-Kontrolle	TRUE
-o	Ungerade-Zeiger-Kontrolle	FALSE
-t	Typkontrolle	TRUE
-z	Lokale Variablen löschen	TRUE
-m	Kleines Codemodell	FALSE
-d	Kleines Datenmodell	FALSE

-1	Code für MC68010 erzeugen	FALSE
-2	Code für MC68020 erzeugen	FALSE
-3	Code für MC68030 erzeugen	FALSE
-8	Code für FPU MC68881/2 erzeugen	FALSE
-a	Garbage-Collector benutzen	TRUE
-e	Spracherweiterungen benutzen	TRUE
-g	Code für Debugger ODebug	FALSE
-y	Neue Symboldatei erzeugen	TRUE
-i	Piktogramme erzeugen	FALSE

Compileroptionen

Name	Opt	Merkmal	Vorgabe	Bezug
StackChk	-s	STACKCHK	TRUE	stap.
OvflChk	-v	OVFLCHK	TRUE	stap.
RangeChk	-b	RANGECHK	TRUE	stap.
CaseChk	-c	CASECHK	TRUE	stap.
ReturnChk	-r	RETURNCHK	TRUE	stap.
NilChk	-n	NILCHK	TRUE	stap.
OddChk	-o	ODDCHK	FALSE	stap.
TypeChk	-t	TYPECHK	TRUE	stap.
ClearVars	-z	CLEARVARS	TRUE	stap.
Debug	-g	DEBUG	FALSE	stap.
EntryExitCode			TRUE	Proz.
CopyArrays			TRUE	Proz.
SaveRegs			FALSE	Proz.
SaveAllRegs			FALSE	Proz.
DeallocPars			TRUE	Proz.
Implementation			TRUE	Modul
CodeChip			FALSE	Modul
VarsChip			FALSE	Modul
DataChip			FALSE	Modul

Bei bedingter Compilation abfragbare Optionen

Name	Opt	Merkmal	Vorgabe
ClearVars	-z	CLEARVARS	TRUE
SmallCode	-m	SMALLCODE	FALSE
SmallData	-d	SMALLDATA	FALSE
MC68010	-1	MC68010	FALSE
MC68020	-2	MC68020	FALSE
MC68030	-3	MC68030	FALSE
FPU	-8	MC68881	FALSE
GarbageCollector	-a	GARBAGECOLLECTOR	TRUE
Extensions	-e	EXTENSIONS	TRUE
Debug	-g	DEBUG	FALSE
EntryExitCode			TRUE
Implementation			TRUE

Steuerzeichen in Zeichenkettenkonstanten

Zeichen	Bedeutung	Wert
\n	line feed	0AX
\t	tabulator	09X
\r	carriage return	0DX
\b	back space	08X
\f	form feed	0CX
\o	nul	00X
\e	escape	1BX
\\	backslash	5CX
\'	single quote	"'"
\"	double quote	'"'
\NNN	Zeichen mit Octalwert NNN	
\xHH	Zeichen mit Hexadezimalwert HH	0HHX
\[	Control Sequence Introducer CSI	9BX

Operatoren und Spezialsymbole

"	(	,	/	<	>=	{
#	)	-	:	<=	[	|
&	*	.	:=	=	]	}
'	+	..	;	>	^	~

Reservierte Wörter

AND	ELSE	MOD	STRUCT
ARRAY	ELSIF	MODULE	THEN
BEGIN	END	NOT	TO
BPOINTER	EXIT	OF	TYPE
BY	FOR	OR	UNTIL
CASE	IF	POINTER	UNTRACED
CLOSE	IMPORT	PROCEDURE	VAR
CONST	IN	RECORD	WHILE
DIV	IS	REPEAT	WITH
DO	LOOP	RETURN	

Standardbezeichner

ABS	ENTIER	LONGINT	REAL
ASH	EXCL	LONGREAL	SET
BOOLEAN	FALSE	LONGSET	SHORT
CAP	HALT	MAX	SHORTINT
CHAR	INC	MIN	SHORTSET
CHR	INCL	NEW	SIZE
COPY	INTEGER	NIL	TRUE
DEC	LEN	ODD	
DISPOSE	LONG	ORD	

Wertebereiche

Typ	MIN(Typ)	MAX(Typ)
SHORTINT	-128	127
INTEGER	-32768	32767
LONGINT	-2147483648	2147483647
REAL	-9.22337177E+18	9.22337177E+18
LONGREAL	-1.7976931348E+308	1.7976931348E+308
SHORTSET	0	7
SET	0	15
LONGSET	0	31
BOOLEAN	FALSE	TRUE
CHAR	0X	0FFX
SYSTEM.BYTE	0X	0FFX

Anhang D: Umstieg von Amiga Oberon 2.14 auf Amiga Oberon 3.0

D

Die vielen Verbesserungen und Erweiterungen von Amiga Oberon in der Version 3.0 machen es leider nötig, daß manche mit Amiga Oberon 2.14 geschriebenen Module nur nach ein paar Änderungen übersetzt werden können. Betroffen sind hier systemnah geschriebene Module oder auch Module, die Anweisungen in Assembler oder anderen Programmiersprachen enthalten. Die wichtigsten Änderungen sind die im folgenden beschriebenen:

Zeigertypen

Durch die Speicherverwaltung mit einem Garbage-Collector müssen nun alle Zeigervariablen auf Objekte zeigen, die von Garbage-Collector angefordert wurden, also nur solche Objekte, die mit der Standardprozedur *NEW* erzeugt wurden. Eine Zuweisung eines anderen Wertes als eines Zeigers auf ein Objekt des Garbage-Collectors an eine Zeigervariable des Typs *POINTER TO T* (wobei *T* ein beliebiger Typ ist) führt zu einem Absturz des Garbage-Collectors!

Auf Systemstrukturen und auf Strukturen in anderen Programmiersprachen geschriebener Programme kann mit gewöhnlichen Zeigern nicht zugegriffen werden. Stattdessen muß der Typ *UNTRACED POINTER TO T* verwendet werden. Mit diesem Typ hat man die Freiheiten der gewöhnlichen Zeiger von Amiga Oberon 2.14.

Variablen der Typen *POINTER TO T* und *UNTRACED POINTER TO T* sind nicht zuweisungskompatibel und dürfen nicht gemischt verwendet werden.

Die Variable *MemReqs* aus *OberonLib* bestimmt nur den Typ des Speichers, der für Zeigervariablen des Typ *UNTRACED POINTER*

TO T mit *NEW* angefordert wird. Der Typ des Speichers für gewöhnliche Zeigervariablen kann vom Programm aus nicht beeinflußt werden.

Der Typ BYTE

BYTE ist bei Amiga Oberon 3.0 kein Standardbezeichner mehr, sondern muß aus dem compilerinternen Modul *SYSTEM* importiert werden.

VAR-Parameter des Typs *ARRAY OF BYTE* müssen mit großer Vorsicht behandelt werden, wenn ihnen Variablen eines verfolgten Typs, also Variablen die gewöhnliche Zeiger enthalten, zugewiesen werden. Die Zeigervariablen dürfen hier nicht verändert werden.

Offene Feldparameter

Offene Feldparameter dürfen bei Amiga Oberon 3.0 beliebig groß sein. Bei der Version 2.14 waren sie auf 32767 Bytes beschränkt.

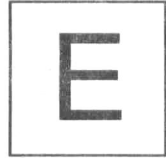
Durch diese Änderung ist der Ergebnistyp der Standardprozedur *LEN* nun vom Typ *LONGINT* und nicht mehr *INTEGER*. Programme, die mit offenen Feldern arbeiten, müssen beim Aufruf von *LEN* also evtl. angepaßt werden. Ist die Beschränkung auf maximal 32767 Feldelemente tragbar, so können in den betroffenen Programmen alle Aufrufe von *LEN(x)*, wobei *x* ein offener Feldparameter ist, durch *SHORT(LEN(x))* ersetzt werden.

Durch die Aufhebung der Größenbeschränkung der offenen Feldparameter hat sich auch das interne Format geändert, mit dem diese Parameter übergeben werden (Kapitel 18). Dadurch funktionieren Assemblerroutinen, die von dem alten Format ausgehen, mit Amiga Oberon 3.0 nicht mehr. Ein Beispiel für eine solche Routine ist die Prozedur *Length* des Moduls *Strings*. Programme, die diese oder ähnliche Prozeduren enthalten, müssen geändert werden.

Recordtypen

Der Typ der Erweiterung einer Recordvariable wird nun nicht mehr als letztes Recordelement in jeder Erweiterungsstufe gespeichert, sondern das erste Recordelement enthält einen Zeiger auf einen Typdescriptor (Kapitel 18). Dadurch werden Recordvariablen kleiner. Da sich hierbei die Adressen der Recordelemente verändern können, kann es zu Inkompatibilitäten kommen.

Anhang E: Literaturverzeichnis



Verweise auf weiterführende Literatur sind im Text durch eckige Klammern und einem Kürzel des Buchtitels gekennzeichnet. Das Kürzel besteht meist aus dem Nachnamen des Autors und dem Erscheinungsjahr. Auf die folgenden Veröffentlichungen wird im Text verwiesen:

[AmigaDos 92]

The AmigaDOS Manual, 3rd Edition
Commodore Amiga, Inc., 1991
Bantam Computer Books, Toronto
ISBN 0-553-35403-5

[AMOK]

Public-Domain-Disketten des Amiga Modula und Oberon Klubs Stuttgart. Zu beziehen sind diese Disketten über jeden guten PD-Vertrieb und über die
A+L AG
Däderiz 61
CH-2540 Grenchen

[digital 88]

Modula-3 Report
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson 1988
digital, Systems Research Center
Palo Alto, California, USA

[Dijkstra 78]

On-the-fly Garbage Collection: An Exercise in Cooperation
Edsger W. Dijkstra
Communications of the ACM November 1978
Volume 21, Number 11

[Fish]

Die bekannteste Public-Domain-Diskettenserie für den Amiga wird von Fred Fish zusammengestellt. Die Disketten können bei fast alle PD-Vertrieben erworben werden. Hier sei auf die Anzeigen in der aktuellen Fachpresse verwiesen. Es lohnt sich dabei oft, die Preise zu vergleichen.

[Meyer 92]

Eiffel: The Language
Bertrand Meyer, 1992
Prentice Hall International (UK) Ltd
Hertforshire, UK
ISBN 0-13-247925-7

[Mössenböck 91]

The Programming Language Oberon-2
H. Mössenböck, N.Wirth Januar 1992
Institut für Computersysteme
ETH-Zentrum, Zürich

[Reiser 92]

Programming in Oberon - Steps beyond Pascal and Modula
Martin Reiser, Niklaus Wirth, 1992
Addison Wesley Publishing Company, Inc.
Reading, Massachusetts
ISBN 0-201-56543-9

[RKM: Autodocs 91]

Amiga ROM Kernel Reference Manual Includes And Autodocs
Amiga technical reference series, 3rd Edition, 1991
Commodore Amiga, Inc.
Addison Wesley Publishing Company, Inc.
Reading, Massachusetts
ISBN 0-201-56773-3

[RKM: Devices 91]
Amiga ROM Kernel Reference Manual DEVICES
Amiga technical reference series, 3rd Edition, 1991
Commodore Amiga, Inc.
Addison Wesley Publishing Company, Inc.
Reading, Massachusetts
ISBN 0-201-56775-X

[RKM: Hardware 9 1]
Amiga ROM Kernel Reference Manual HARDWARE
Amiga technical reference series, 3rd Edition, 1991
Commodore Amiga, Inc.
Addison Wesley Publishing Company, Inc.
Reading, Massachusetts
ISBN 0-201-56776-8

[RKM: Libraries 92]
Amiga ROM Kernel Reference Manual LIBRARIES
Amiga technical reference series, 3rd Edition, 1992
Commodore Amiga, Inc.
Addison Wesley Publishing Company, Inc.
Reading, Massachusetts
ISBN 0-201-56774-1

[Style 91]
Amiga User Interface Style Guide
Amiga technical reference series, 1991
Commodore Amiga, Inc.
Addison Wesley Publishing Company, Inc.
Reading, Massachusetts
ISBN 0-201-57757-7

[Williams 89]
68030 Assembly Language Reference - Including the 68020
Steve Williams, 1989
Addison Wesley Publishing Company, Inc.

Reading, Massachusetts
ISBN 0-201-08876-2

[Wirth 83]
Algorithmen und Datenstrukturen
Niklaus Wirth, 1983
Verlag B.G. Teubner, Stuttgart
ISBN 3-519-02250-8

[Wirth 85]
Programming in Modula-2, 3rd Edition
Niklaus Wirth, 1985
Springer Verlag
Berlin, Heidelberg
ISBN 3-540-13301-1

[Wirth 87]
From Modula to Oberon and
The Programming Language Oberon
Niklaus Wirth, 1987
Institut für Informatik, Fachgruppe Computer-System
ETH-Zentrum, Zürich

[Wirth 90]
The Programming Language Oberon (Revision 1. 10. 90)
Niklaus Wirth, 1990
Institut für Informatik, Fachgruppe Computer-System
ETH-Zentrum, Zürich

Index

A

B

P

17. Der Debugger ODebug

Dieser Runtime-Source-Level-Debugger ist ein nützliches Werkzeug beim Entwickeln von Programmen mit dem Amiga Oberon System. Er verschafft Anfängern Einblicke in den Ablauf von Programmen und ist für fortgeschrittene Software-Entwickler unverzichtbar.

Durch das Arbeiten mit diesem Debugger hat man den Komfort, den man sonst nur von Interpretersprachen gewohnt ist, auch bei der Entwicklung von compilierten Oberon-Programmen mit dem Amiga Oberon Compiler. Dieser Debugger hat eine Vielzahl von Funktionen die z.B. das schrittweise Ausführen eines Programms, das Auslesen der Werte globaler und lokaler Variablen oder das Setzen von sogenannten Breakpoints auf beliebige Positionen im Quelltext erlauben.

Um ODebug verwenden zu können, ist Amiga Oberon V2.0 nötig. Sollten Sie diese Version noch nicht besitzen, können Sie das Update günstig bei der A+L AG beziehen.

Hardwareanforderungen:

Um sinnvoll mit ODebug arbeiten zu können, sollte man einen Amiga mit mindestens 1 MByte freiem Speicher und 2 Diskettenlaufwerken besitzen. Beim Entwickeln größerer Programme, die aus vielen Modulen bestehen, ist mehr Speicher ratsam.

ODebug wurde für die Oberfläche das AmigaOS 2.0 gestaltet. Er kann jedoch auch problemlos mit AmigaOS 1.2 bzw. 1.3 arbeiten, mit AmigaOS 2.0 sieht er jedoch ansprechender aus.

Installation:

Haben Sie den Amiga Oberon Compiler bereits auf Ihrer Harddisk installiert, reicht es, wenn Sie InstallHD der ODebug-Diskette durch

Doppelklick starten. Es kopiert den Debugger in das Verzeichnis OBERON: und die nötigen Objekt- und Symboldateien nach OBERON:OBJ/ bzw. OBERON:SYM/.

Arbeiten Sie mit Diskettenlaufwerken, sollten sie die Dateien aus den SYM- und OBJ-Verzeichnissen der ODebug-Diskette zu Ihren anderen Objekt- und Symboldateien kopieren. Wenn Sie dies nicht möchten, reicht es auch, die Pfadliste OBERON:Path um den Eintrag ODebug: zu erweitern. Möchten Sie von der Workbench-Oberfläche aus arbeiten, sollten Sie den Inhalt des Icons-Verzeichnisses der ODebug-Diskette zu den anderen Oberon-Icons in OBERON:Icons/ kopieren.

Start des Debuggers:

Alle Module eines Programms, die eventuell fehlerhaft sind und daher im Debugger angezeigt werden sollen, müssen zunächst mit der Compileroption '-g' compiliert werden. Wird vom Editor OEd aus compiliert, muß im Menü *Options* der Menüpunkt *Debug* angewählt sein. Bei der Compilation wird automatisch das Modul 'Debug' importiert und es wird spezieller Code erzeugt, damit die compilierten Module im Debugger angezeigt werden können. Außerdem erzeugt der Compiler Referenzdateien (mit der Endung '.ref'), die zusätzliche Informationen über die Module enthalten, und ohne die ODebug nicht arbeiten kann. Existiert im aktuellen Verzeichnis ein Unterverzeichnis mit dem Namen 'ref', werden die Referenzdateien dort gespeichert.

Nach der Compilation dieser Module muß das Programm noch wie ein normales Oberon-Programm mit OLink gelinkt werden.

Start vom CLI:

Vor dem Start von ODebug sollte die Größe des Stapelspeichers wie beim Compiler mit 'stack 20000' hochgesetzt werden. Der Debugger sollte vom CLI aus nur mit dem Befehl *run* gestartet werden, da man

sonst, solange der Debugger aktiv ist, ein 'totes' CLI-Fenster übrig behält. Beim Start können, ähnlich wie beim Editor OEd, folgende Argumente übergeben werden:

Aufruf:

run ODebug {-{x#|y#|w#|h#|d#|s|l|p|r|c|o}}

Dabei steht das '#' für eine Dezimalzahl. Die Optionen haben folgende Bedeutung:

-x#, -y#	x- und y-Position der ODebug-Fenster
-w#, -h#	Breite und Höhe der ODebug-Fenster
-s	ODebug soll einen eigenen Screen öffnen
-l	Screen soll im Interlace-Modus geöffnet werden
-d#	Tiefe des ODebug-Screens, voreingestellt ist 2
-p	'Popup Windows' deaktivieren
-r	'Sort Records' aktivieren
-c	'Close Pointers' aktivieren
-o	Ist diese Option gesetzt, öffnet ODebug die Fenster, in denen Quelltexte angezeigt werden, nicht alle automatisch beim Programmstart.

Genauere Informationen zu den letzen vier Parametern stehen in dem Abschnitt über das *Options*-Menü.

Start von der Workbench:

Von der Workbench wird ODebug einfach durch einen Doppelklick auf sein Icon gestartet. Parameter können über sogenannte Tool Types angegeben werden. Diese können nach dem Anwählen des ODebug-Icons mit dem Menüpunkt 'Info' des Workbench-Menüs geändert werden.

Folgende Tool Types können angegeben werden:

LEFTEDGE	x-Position der ODebug-Fenster
TOPEDGE	y-Position der ODebug-Fenster
WIDTH	Breite der ODebug-Fenster
HEIGHT	Höhe der ODebug-Fenster
SCREEN	Soll ein Screen geöffnet werden ('TRUE') oder nicht ('FALSE')
INTERLACE	Screen im Interlace-Modus ('TRUE') oder im Non-Interlaced-Modus ('FALSE') öffnen
DEPTH	Tiefe des ODebug-Screens
POPUPWINDOWS	Option 'Popup Windows' setzen ('TRUE') oder löschen ('FALSE')
SORTRECORDS	Option 'Sort Records' setzen ('TRUE') oder löschen ('FALSE')
CLOSEPOINTERS	Option 'Close Pointers' setzen ('TRUE') oder löschen ('FALSE')
OPENSOURCES	Quelltext-Fenster beim Programmstart automatisch öffnen ('TRUE'), oder nur die benötigten Quelltexte anzeigen ('FALSE').

Die letzen vier Tool Types betreffen direkt die Voreinstellungen des *Options*-Menü, das weiter unten beschrieben wird.

Arbeitsweise von ODebug:

Nach dem Start von ODebug wird ein schmales Fenster mit der Meldung 'Bitte mit Option '-g' compiliertes Programm starten!' geöffnet. Sie sollten nun also das zuvor compilierte und gelinkte Programm starten. Dabei können Sie es, je nach Belieben, von der Workbench oder vom CLI, mit oder ohne Parameter, starten. Der Debugger

schließt daraufhin sein Fenster und öffnet für jedes Modul, das zuvor mit der Option '-g' compiliert wurde, ein Fenster, des den Quelltext dieses Moduls enthält. Die erste Anweisung, die Ihr Programm ausführen möchte, wird hervorgehoben dargestellt. Sie haben nun über das Menü des Debuggers volle Kontrolle über den Ablauf des Programms und können es schrittweise ablaufen lassen, oder auch über längere Programmteile 'rennen' lassen, bis von Ihnen vorgegebene Bedingungen erfüllt sind. Dies können an bestimmten Stellen im Programm gesetzte sogenannte Breakpoints sein, oder auch Variablen, die einen bestimmten Wert annehmen oder gar ein Laufzeitfehler.

Sollten Sie jetzt noch kein Programm debuggen wollen, können Sie den Debugger auch einfach durch Anklicken des Schließ-Gadgets wieder verlassen.

Die von ODebug geöffneten Fenster besitzen an ihren rechten und unteren Rändern Proportional- und Pfeil-Gadgets, mit denen man sich wie bei Workbench-Fenstern durch die angezeigten Informationen bewegen kann. Dies funktioniert bei allen Fenstern von ODebug gleichermaßen.

Die Fenster können durch Anwahl ihres Close-Gadgets geschlossen werden. Es muß jedoch mindestens ein Fenster, in dem der Quelltext eines Moduls angezeigt wird, geöffnet bleiben. Wird auch bei dem letzten Quelltext-Fenster das Close-Gadget angewählt, wird dies wie das Anwählen von *Abort Debug* des *Project*-Menüs behandelt (siehe unten).

Durch Anwahl des Close-Gadgets geschlossene Quelltext-Fenster werden automatisch wieder geöffnet und angezeigt, sobald das debuggte Programm Anweisungen des Moduls, dessen Quelltext in dem Fenster angezeigt wurde, ausführt. Dies geschieht z.B. beim Aufruf einer Prozedur dieses Moduls aus einem anderen Modul.

Die Menüs:

Das Projekt Menü:

Abort Debug (Amiga + 'A'):

Das Programm, das sich im Augenblick im Debugger befindet, wird abgebrochen und beendet. Der Debugger gibt den für Quelltexte und Referenzdateien allozierten Speicher frei und schließt seine Fenster. Danach öffnet er wieder das schmale Fenster und wartet, bis das nächste zu debuggende Programm gestartet wird oder der Debugger beendet wird.

Wird im Augenblick ein Progamm debuggt, das keine korrekten CLOSE-Anweisungen besitzt und nicht alle Resourcen, wie geöffnete Fenster oder allozierten Speicher, sauber zurückgibt, werden diese auch beim Anwählen von *Abort Debug* nicht freigegeben.

Enthält das debuggte Programm schwere Fehler im CLOSE-Teil, kann es passieren, daß es bei *Abort Debug* abstürzt.

Quit (Amiga + 'Q'):

Quit beendet das Programm, das sich gerade im Debugger befindet, so wie es bei *Abort Debug* geschieht. Danach wird jedoch auch der Debugger selbst beendet. Um ein weiteres Programm zu debuggen muß ODebug also neu gestartet werden.

About:

Es wird ein Fenster mit Informationen über die Version und den Autor des Debuggers geöffnet.

Das Debug Menü:

Step (Amiga + 'S', Leertaste):

Step führt die zur Zeit hervorgehobene Anweisung des debuggten Programms aus. Die logisch folgende Anweisung wird hervorgehoben dargestellt. Sie kann danach durch nochmaliges Anwählen von *Step* ausgeführt werden. War die ausgeführte Anweisung ein Prozeduraufruf, ist die folgende Anweisung die erste Anweisung der aufgerufenen Prozedur. Es wird daher auch im Debugger diese Prozedur angezeigt.

Mit dieser Funktion kann man sich Anweisung für Anweisung durch ein Programm hangeln und so Fehler oder ungeplante Irrwege, in die ein Programm läuft, aufspüren.

Walk (Amiga + 'W'):

Mit *Walk* kann ein Programmablauf ähnlich wie bei *Step* verfolgt werden. Hier werden die Anweisungen jedoch automatisch nacheinander ausgeführt und angezeigt. Zwischen den Anweisungen macht der Debugger jeweils 0.2 Sekunden Halt, so daß man dem Programmlauf leicht folgen kann.

Abgebrochen wird *Walk* indem man mit der linken Maustaste in ein Fenster des Debuggers klickt oder einen Menüpunkt anwählt.

Außerdem wird beim Erreichen eines sogenannten Breakpoints (siehe nächstes Menü) oder beim Auftreten eines Laufzeitfehlers abgebrochen. In beiden Fällen wird ein Requester geöffnet, der über die Ursache des Abbruchs Auskunft gibt.

Run (Amiga + 'R'):

Run startet das debuggte Programm an der aktuellen Anweisung. Der Programmablauf kann dabei nicht am Bildschirm verfolgt werden.

Abgebrochen wird *Run* wie *Walk* durch einen Druck auf den linken Mausknopf oder Anwählen eines Menüpunktes.

Breakpoints und Laufzeitfehler stoppen das mit *Run* gestartete Programm ähnlich wie bei *Walk*.

Execute (Amiga + 'X'):

Execute führt die hervorgehobene Anweisung ähnlich wie *Step* aus. Dabei wird die Ausführung aber erst beim Erreichen der physikalisch nächsten, d.h. der im Quelltext folgenden, Anweisung abgebrochen. Dies ist nützlich, wenn die hervorgehobene Anweisung ein Prozeduraufruf ist, man sich jedoch nicht durch die Prozedur hangeln möchte. Ist die nächste Anweisung eine Schleife, wird sie bei *Execute* fertig ausgeführt und man muß nicht jede einzelne Schleifenanweisung und jeden Schleifendurchlauf einzeln verfolgen.

Execute kann auch durch die linke Maustaste oder durch Anwahl eines Menüpunktes vorzeitig gestoppt werden. Beim Erreichen eines Breakpoints oder beim Auftreten eines Laufzeitfehlers wird auch hier die Ausführung abgebrochen.

Run Quick (Amiga + 'K'):

Ein mit *Run* gestartetes Programm läuft oft sehr langsam. Mit *Run Quick* kann man ein Programm schneller bis zu einem Breakpoint oder einem Laufzeitfehler laufen lassen.

Die Nachteile von *Run Quick* sind zum einen, daß das Programm nicht abgebrochen werden kann, was bei *Run* durch einen einfachen Mausklick jederzeit möglich ist. Zudem ignoriert *Run Quick* die Breakpoint-Ausdrücke, die mit *Add Expression* im *Breakpoint*-Menü eingegeben werden können.

Run Quick sollte z.B. zum schnellen Durcharbeiten von aufwendigen Initialisierungteilen eines Programms verwendet werden. Man sollte

jedoch nicht vergessen, zuvor Breakpoints zu setzen, damit das debuggte Programm die Kontrolle wieder an den Debugger zurück gibt.

Sprint (Amiga + 'I'):

Mit Sprint können schnell Anweisungen gefunden werden, die Laufzeitfehler verursachen.

Durch die vielen Überprüfungen, die während des Laufes eines mit *Run* gestarteten Programms gemacht werden müssen, wird das debuggte Programm eventuell sehr langsam. Mit *Sprint* kann das Programm mit einer Geschwindigkeit laufen, die vergleichbar ist mit der Geschwindigkeit, mit der das Programm ohne Debugger laufen würde. *Sprint* wird jedoch nur abgebrochen, wenn beim debuggten Programm ein Laufzeitfehler auftritt. Danach wird die Anweisung, die zum Laufzeitfehler führte, hervorgehoben angezeigt. Nach dem Start mit *Sprint* können nur die Inhalte von globalen, nicht jedoch von lokalen Variablen angesehen werden. Die Aufrufsequenz der Prozeduren kann nicht betrachtet werden (siehe Menü *Variables*).

Ein Programm, das in einer Endlosschleife hängen bleibt, sollte nicht mit *Sprint* gestartet werden, da es dann normalerweise nicht mehr abgebrochen werden kann.

Show Source (Amiga + 'H'):

Die Fenster, in denen die Quelltexte der debuggten Module angezeigt werden, können mit dem Close-Gadget geschlossen werden. Außerdem kann durch die Option '-o' oder das Tool Type "OPEN-SOURCES" beim Start von ODebug verhindert werden, daß alle Quelltexte angezeigt werden.

Möchte man nun einen im Augenblick nicht angezeigten Quelltext sehen, wählt man *Show Source*. Es wird dann zunächst ein Fenster geöffnet, in dem der Name eines Moduls durch Doppelklick oder durch Anwahl des Namens und Klick auf 'ok' gewählt werden muß.

Das entsprechende Fenster wird nun geöffnet. Ist dieses Fenster bereits offen, wird es aktiviert und in den Vordergrund geholt.

Show Statement (Amiga + 'N'):

Mit den Gadgets im Rahmen der Quelltext-Fenster kann man sich durch die Quelltexte bewegen. Vergißt man dabei die Position der hervorgehobenen Anweisung, oder möchte man schnell zu dieser zurückspringen, kann man *Show Statement* anwählen.

Ist das Fenster des Quelltextes, der die vorgehobene Anweisung enthält, derzeit nicht geöffnet, wird es geöffnet, da die Anweisung sonst nicht angezeigt werden könnte.

Das Breakpoint Menü:

Eine als Breakpoint markierte Anweisung im debuggten Programm bricht das mit *Walk*, *Run* oder *Execute* gestartete Programm ab, sobald sie erreicht wird. Dabei wird jeweils abgebrochen, bevor die als Breakpoint markierte Anweisung ausgeführt wird. Breakpoints werden im Quelltext durch Fettschrift angezeigt.

Zusätzlich zu normalen Breakpoints erlaubt dieser Compiler auch noch beliebige boole'sche Ausdrücke, die als Abbruchkriterium benutzt werden können.

Set (Amiga + 'T'):

Mit *Set* kann ein Breakpoint gesetzt werden. Zuvor muß eine Anweisung mit der Maus selektiert werden. Dies geschieht einfach durch Anwählen einer Anweisung im Quelltext mit der linken Maustaste. Dabei ist zu beachten, daß Anweisungen ineinander geschachtelt sein können. So besteht die Programmzeile

```
IF a>b THEN a := b END;
```

aus den Anweisungen 'a := b' und der gesamten IF-Anweisung. Würde ein Breakpoint auf 'a := b' gesetzt, wird das Programm nur abgebrochen, wenn die Bedingung 'a>b' erfüllt ist, während beim Setzen eines Breakpoints auf die gesamte IF-Anweisung jedesmal beim Erreichen dieser abgebrochen wird.

Außer auf Anweisungen können Breakpoints auch auf die Schlüsselwörter END, CLOSE, UNTIL, ELSE, ELSIF und auf '|' gesetzt werden. So kann man eine Prozedur z.B. nach der Ausführung ihrer letzten Anweisung abbrechen, um sich die Werte ihrer lokalen Varialben beim Verlassen der Prozedur anzusehen (siehe Menü *Variables*).

Bei der IF-Anweisung oben würde ein Breakpoint auf dem Schlüsselwort END das Programm zum Abbruch bringen, nachdem 'a := b' ausgeführt wurde.

Remove (Amiga + 'M'):

Um einen mit *Set* gesetzten Breakpoint wieder zu entfernen, selektiert man ihn zunächst mit der linken Maustaste und wählt dann *Remove*.

Add Expression (Amiga + 'D'):

Mit diesem Menüpunkt können boole'sche Ausdrücke, die zum Abbruch des Programms führen sollen, eingegeben werden. Es wird zunächst ein Fenster mit einem Text-Gadget geöffnet. Hier kann ein beliebiger Ausdruck, wie er auch in Oberon-Programmen vorkommen kann, eingegeben werden. Man kann dabei auf alle im debuggten Modul und von diesem importierten Modulen sichtbare Variablen und Konstanten zugreifen und damit rechnen. Das Ergebnis des Ausdrucks muß vom Typ BOOLEAN sein, wobei der Wert TRUE zum Abbruch des Programms führt.

Beispiel:

(x>Window.width) OR (y>Graphics.gfx.normalDisplayRows)

Dieser Ausdruck würde das Programm abbrechen, sobald die Variable *x* größer als die Breite des Fensters, auf das *Window* zeigt oder *y* größer als die normale Bildhöhe, wie sie von der graphics.library vorgegeben wird, ist. Natürlich müssen die Variablen *x*, *y* und *Window* im debuggten Modul definiert und das Modul *Graphics* importiert sein.

Durch Drücken von Return im Text-Gadget wird der derzeitige Wert des Ausdrucks angezeigt.

Mit dem Gadget unter dem Text-Gadget kann gewählt werden, ob ein Fehler, der beim Auswerten des Ausdrucks auftreten kann, auch zum Abbruch führen soll. Ist 'Abbruch bei Fehler' angewählt, würde der Ausdruck oben z.B. auch zum Abbruch führen, wenn *Window=NIL* ist. Genauso würde abgebrochen, wenn *x* und *y* lokale Variablen sind, und eine Prozedur aufgerufen wird, von der aus diese Variablen nicht mehr sichtbar sind.

Durch Anwahl von 'ok' wird der Ausdruck als neue Abbruchbedingung übernommen. Es können beliebig viele Abbruchbedingungen eingegeben werden.

Mit 'Abbruch' kann das Fenster verlassen werden, ohne die Liste der Abbruchbedingungen zu ändern.

Nach dem Abbruch des Programms durch eine Abbruchbedingung wird dieses Fenster wieder geöffnet. Es enthält genauere Informationen über den Grund des Abbruchs und erlaubt es, die Abbruchbedingung aus der Liste zu entfernen (durch Anwählen von 'löschen') oder sie beizubehalten ('Abbruch').

Change Expression (Amiga + 'J'):

Dieser Menüpunkt erlaubt es, zuvor mit *Add Expression* eingegebene Abbruchbedingungen zu verändern. Dazu wird zunächst ein Fenster geöffnet, in dem man eine Abbruchbedingung anwählt. Diese kann dann ähnlich wie bei *Add Expression* verändert werden.

Remove Expression (Amiga + 'O'):

Diese Funktion erlaubt es, mit *Add Expression* eingegebene Abbruchbedingungen wieder zu löschen. Dazu muß die Bedingung in einem Fenster angewählt werden.

Clear All (Amiga + 'C'):

Dieser Menüpunkt entfernt alle Breakpoints und Abbruchbedingungen.

Das Variables Menü:

Show (Amiga + 'V'):

Mit *Show* kann man sich die Werte der Variablen eines Sichtbarkeitsbereichs (eines Moduls oder einer Prozedur) ansehen. Dazu wird ein Fenster geöffnet, in dem man einen Sichtbarkeitsbereich wählen kann. Diese werden mit den Namen der Module, in denen sie definiert wurden, bezeichnet. Die globalen Variablen des aktuellen Moduls heißen 'Globals: Modulname', die lokalen Variablen einer aktiven Prozedur 'Locals: Prozedurname'.

Nach der Wahl des Sichtbarkeitsbereichs wird ein Fenster geöffnet, das in jeder Zeile den Namen, den Wert und in eckigen Klammern den Typbezeichner einer Variablen enthält. Die Werte von strukturierten Variablen (Felder und Records) können nicht in einer Zeile angezeigt werden, stattdessen steht dort 'ARRAY ...' bzw. 'RECORD ...'. Durch Doppelklick auf eine solche Zeile wird ein neues Fenster geöffnet, das

den Inhalt der Feld- bzw. Recordelemente enthält. Entsprechend kann man sich die Werte, auf die ein Zeiger zeigt, ansehen. Alle Variablen, deren Werte durch Doppelklick angezeigt werden können, sind durch drei Punkte ('...') gekennzeichnet. Auf diese Weise kann man sich leicht durch kompliziertere Strukturen wie Bäume oder Listen durcharbeiten. Enthält ein Zeiger NIL oder einen ungeraden Wert, wird zur Warnung ein Requester geöffnet.

Nicht mehr benötigte Fenster, in denen die Werte von Variablen angezeigt werden, können durch Anklicken des Close-Gadgets geschlossen werden.

Extend (Amiga + 'E'):

Dieser Menüpunkt kann nur angewählt werden, wenn im aktiven Fenster der Inhalt eines RECORDs angezeigt wird. Es wird geprüft, ob der RECORD erweitert wurde. Ist dies der Fall, werden die nächste Erweiterung und die in der Erweiterung neu hinzugekommenen Elemente angezeigt. Ist der RECORD nicht erweitert oder ist der Typ der Erweiterung vom aktuellen Modul aus nicht sichtbar, wird eine entsprechende Meldung ausgegeben.

Change Value (Amiga + 'G'):

Change Value erlaubt das Ändern des Wertes einer vorher angewählten Variablen. Es kann ein beliebiger Ausdruck eingegeben werden, dessen Ergebnis der Variablen zugewiesen wird. Das Ergebnis des Ausdrucks muß natürlich zuweisungskompatibel zur angwählten Variablen sein.

Diese Funktion sollte nur sehr vorsichtig verwendet werden, da von außen unerwartet geänderte Variablenwerte viele Programme leicht durcheinander bringen.

Evaluate Expression (Amiga + 'U'):

Dieser Menüpunkt erlaubt es, beliebige Ausdrücke, in denen im debuggten Modul definierte Variablen und Konstanten vorkommen, zu berechnen. Die Schreibweise des Ausdrucks entspricht der normaler Ausdrücke, wie sie auch in Oberon-Programmen vorkommen können, lediglich Prozeduraufrufe von anderen als den Standardprozeduren sind verboten. Es kann mit allen von Oberon bekannten Typen (INTEGER, SET, REAL, BOOLEAN, etc.) gerechnet werden. Das Ergebnis und der Typ des Ergebnisses wird angezeigt.

Beispiele:

```
a + b * 3 MOD 45 - ENTIER(a * 3.124 / b)
(b IN (set1 * set2)) OR (node IS ExtendedNode)
ODD(Intuition.int.activeScreen.rastport.bitMap.bytesPerRow)
```

Wird ein fehlerhafter Ausdruck eingegeben, wird eine entsprechende Fehlermeldung ausgegeben.

Diese Funktion kann mit dem 'Abbruch'-Gadget wieder verlassen werden.

Change Type (Amiga + 'Y'):

Es ist möglich, den Typ, mit dem eine Variable angezeigt wird, zu verändern. Dies funktioniert ähnlich wie bei der Prozedur VAL() des Moduls SYSTEM, der neue Typ sollte also die gleiche Größe des ursprünglichen Typs haben. Diese Funktion betrifft das debuggte Programm nicht, es wird lediglich der Wert der Variablen angezeigt, als wäre sie von einem anderen Typ.

Um einen neuen Typ zu wählen muß zunächst der Sichtbarkeitsbereich des Typbezeichners gewählt werden. Dabei beinhaltet 'Standardbezeichner' die Standardtypen. Nun muß der Typ selbst noch gewählt werden.

Show active Procedures (Amiga + 'P'):

Diese Funktion öffnet ein Fenster in dem die Namen aller derzeitig aktiven Prozeduren angezeigt werden. Dies sind diejenigen Prozeduren, deren lokale Variablen sichtbar sind.

Show called Procedures (Amiga + 'L'):

Diese Funktion öffnet ein Fenster in dem die Namen aller derzeitig aufgerufenen Prozeduren in der Reihenfolge ihres Aufrufs angezeigt werden. So kann z.B. der Verlauf einer Rekursion leicht zurückverfolgt werden.

Save Contents:

Der Inhalt eines Fensters, das Variablenwerte oder Prozedurbezeichner enthält, kann mit dieser Funktion in eine ASCII-Datei gespeichert oder auf dem Drucker ausgegeben werden. Dazu wird zunächst nach dem Dateinamen gefragt. Wird 'PRT:' als Dateiname eingegeben, wird der Inhalt auf dem Drucker ausgegeben. Es wird nicht nur der sichtbare, sondern der gesamte Fensterinhalt gespeichert. Die erzeugte Datei kann mit einem beliebigen Texteditor bearbeitet oder mit dem CLI-Befehl TYPE ausgegeben werden.

Das Options Menü:

Popup Windows (Amiga + '0'):

Ist diese Option angewählt, wird beim Wechsel zwischen zwei Modulen, z.B. wenn in einem Modul eine Prozedur eines anderen Moduls aufgerufen wird, das Fenster des Moduls, in das gesprungen wird, nach vorne geholt. Dies ist vor allem beim gleichzeitigen Debuggen mehrerer Module nützlich.

Sort RECORDs (Amiga + '1'):

Die Inhalte von RECORDs werden alphabetisch sortiert angezeigt, wenn diese Option angewählt ist. Ansonsten werden sie in der Reihenfolge, in der sie definiert wurden, ausgegeben.

Close Pointers (Amiga + '2'):

Normalerweise werden alle Fenster, die die Inhalte von RECORDs oder ARRAYs anzeigen, und die man mit Hilfe von Zeigern erreicht hat, geschlossen, sobald die Kontrolle an das debuggte Programm übergeht (bei *Step*, *Walk*, etc.). Dies ist nötig, da der Inhalt der Fenster ungültig werden kann, wenn z.B. ihr Speicher vom debuggten Programm freigegeben wird.

Ist diese Option nicht angewählt, werden solche Fenster nicht automatisch geschlossen. Dadurch erspart man sich das manchmal lästige erneute Öffnen der Fenster. Man hat jedoch keine Garantie dafür, daß die Fenster noch gültige Werte enthalten.

Open Sources (Amiga + '3'):

Ist diese Option gelöscht, werden beim Start eines Programms, das debuggt werden soll, nicht automatisch die Quelltexte aller beteiligten Module angezeigt. Es werden nur jeweils die Fenster geöffnet, die Anweisungen enthalten, die das Programm ausführt.

Das Abschalten dieser Option ist beim Debuggen von Programmen, die aus vielen Modulen bestehen, wichtig. Bei zu vielen geöffneten Fenstern kann es, bei wenig Chip-Memory, zu Speicherproblemen kommen. Die benötigten Quelltext-Fenster können nachträglich mit der Funktion *Show Source* des *Debug*-Menüs angezeigt werden.

Die Compileroption Debug:

Der Amiga Oberon Compiler kennt ab Version 2.0 die stapelbare Op-

tion $Debug. Mit ihr kann das Erzeugen von Code für den Debugger abschnittweise abgeschaltet werden. Dies geschieht, indem man den betroffenen Abschnitt im Quelltext mit '(* $Debug- *)' und '(* $Debug= *)' umschließt. '(* $Debug+ *)' ist nicht möglich, das Erzeugen von Debug-Code kann nur beim Start des Compilers mit '-g' aktiviert werden.

Das abschnittweise Ausschalten des Debug-Codes ist manchmal nötig, da dieser Code sehr viel langsamer ist. So kann man z.B. Schleifen, die oft durchlaufen werden und mit großer Wahrscheinlichkeit fehlerfrei sind, vom Debuggen ausnehmen. Diese Schleifen werden dann beim Debuggen schnellstmöglich ausgeführt, man hat jedoch keine Möglichkeit, deren Ausführung abzubrechen.

Beim Debuggen sehr großer Module kann es durch den zusätzlich erzeugten Debug-Code zu einer Überschreitung der 32 KByte-Grenze kommen. Dieses Problem kann man umgehen, indem man Teile des Moduls mit '(* $Debug- *)' vom Debuggen ausnimmt.

Weitere Hinweise:

Hardware-Register:

Man sollte sich niemals den Inhalt der Hardware-Register, die im Module Hardware definiert sind, mit *Show Variables* ansehen. Da es Register gibt, die beim Auslesen ihrer Werte Aktionen auslösen, führt dies zu unvorhersehbaren Katastrophen.

Es ist jedoch möglich, sich einzelne Werte mit Hilfe von *Evaluate Expression* anzusehen. So liefert z.B der Ausdruck 'Hardware.custom.vposr' die aktuelle Position des Videostrahls.

Quelltexte:

Damit ODebug den Quelltext eines Programmes korrekt anzeigen kann, dürfen Quelltexte nach der Compilation mit der Option '-g',

nicht verändert werden. Die Referenzdatei enthält Verweise auf Positionen im Quelltext, die bei einem veränderten Text möglicherweise nicht mehr stimmen.

Sollten Sie dennoch den Debugger und das betroffene Programm starten, öffnet ODebug einen warnenden Requester. Arbeitet man dennoch mit dem Debugger, kann es passieren, daß die angezeigten Anweisungen nicht mit den wirklich ausgeführten Anweisungen übereinstimmen.

Versionskonflikte:

ODebug kann nicht arbeiten, wenn es zwischen den Symboldateien der am debuggten Programm beteiligten Module einen Versionskonflikt gibt. Dieser kann z.B. dadurch entstanden sein, daß ein verändertes importiertes Modul neu compiliert wurde.

ODebug bricht in diesem Fall ab und gibt in einem Requester Auskunft über das Problem. In einem solchen Fall sollte das Programm neu compiliert und gelinkt werden, um die Referenz- und Symboldateien auf den aktuellen Stand zu bringen.

Referenzdateien:

Die beim Compilieren mit ODebug erzeugten Referenzdateien werden automatisch in ein Unterverzeichnis 'ref/' kopiert, wenn ein solches Verzeichnis existiert. Da die Dateien nur von ODebug benötigt werden, ist dieses Verzeichnis zu empfehlen, da die Dateien dort weniger stören.

Die Referenzdateien können mit 'delete ref/#?.ref' oder durch Selektieren ihrer Icons und Anwählen von 'Delete' im Workbench-Menü gelöscht werden, sobald sie nicht mehr benötigt werden. Beim Neucompilieren eines Moduls werden automatisch neue Referenzdateien erzeugt. Anders als bei den Symboldateien enthalten die Referenzdateien keine Versionsnummern, so daß eine Neucompilation des gesam-

ten Programms beim Debuggen eines Untermoduls durch das Löschen von Referenzdateien nicht nötig wird.

18. Der Disassembler DecObj:

Mit DecObj können mit dem Amiga Oberon Compiler erzeugte Objektdateien und gelinkte Programme disassembliert werden. Es können jedoch auch andere BLink-kompatible Objektdateien disassembliert werden, also z.B. auch von dem frei kopierbaren Assembler 'a68k' erzeugte Dateien.

Aufruf vom CLI:

Aufruf:

DecObj <Dateiname>

Dabei ist <Dateiname> der Name einer Objektdatei oder eines ausführbaren Programms. Wird kein Argument übergeben, wartet DecObj ähnlich wie der Compiler auf die Eingabe eines Dateinamens.

Aufruf von der Workbench:

Beim Start von der Workbench sollte zunächst das Icon von DecObj angeklickt und danach das Icon der Objektdatei oder des Programms, das disassembliert werden soll, doppelgeklickt werden.

Arbeitsweise von DecObj:

DecObj liest zunächst die Objektdatei oder das Programm ein. Danach wird ein Disassemblerlisting erzeugt, das mit der Endung '.dis' gespeichert wird. Wurde DecObj von der Workbench aus gestartet, wird für das erzeugte Listing auch noch ein Icon erzeugt.

Der Aufbau des Disassemblerlistings entspricht der Hunk-Struktur der ursprünglichen Datei. Genauere Informationen über den Aufbau von Objektdateien und Programmen findet man z.B. in [rb:agb]. Von gro-

ßem Vorteil beim Betrachten der Disassemblerlistings sind natürlich auch gute Assemblerkenntnisse.

Beispiel:

Das Programm

```
MODULE GGT;

IMPORT io;

VAR a,b: INTEGER;

PROCEDURE ggt(a,b: INTEGER): INTEGER;

BEGIN
  LOOP
    IF    a>b THEN DEC(a,b)
    ELSIF b>a THEN DEC(b,a)
    ELSE
      RETURN a
    END;
  END;
END ggt;

BEGIN
  io.WriteString("a = "); io.ReadIntegerOk(a);
  io.WriteString("b = "); io.ReadIntegerOk(b);
  io.WriteString("GGT(a,b) = "); io.WriteInt(ggt(a,b),0); io.WriteLn;
END GGT.
```

wurde ohne Überprüfungscode mit dem kleinen Code-Modell compiliert. Beim Start von 'DecObj GGT.obj' wird folgendes Disassemblerlisting GGT.dis erzeugt:

```
DecObj Amiga Oberon Disassembler listing: GGT.obj
```

Zunächst ein Hunk, der prüft, ob das Programm bereits gestartet wurde (es ist ohne kleines Datenmodell ja nicht reentrant), den Wert von Register A7 sichert und den BEGIN- und danach den CLOSE-Teil des Hauptmoduls (in diesem Fall GGT) anspringt:

```
HUNK_UNIT: GGT
HUNK_NAME: GGT
HUNK_CODE: 00000000  (0)
00000000: 49F900000000          LEA         $00000000,A4
00000006: 4A94                  TST.L       (A4)
00000008: 6704                  BEQ         $0000000E
0000000A: 7014                  MOVEQ       #20,D0
0000000C: 4E75                  RTS
0000000E: 28BC0000001A          MOVE.L      #$0000001A,(A4)
00000014: 294F0004              MOVE.L      A7,4(A4)
00000018: 61FF                  BSR         $00000019
0000001A: 2E7900000004          MOVEA.L     $00000004,A7
00000020: 50F90000000C          ST          $0000000C
00000026: 6100006C              BSR         $00000094
0000002A: 4E75                  RTS
HUNK_RELOC:32
Offsets in Hunk no: 00000000  (0)
00000000: 00000010 ....
HUNK_EXT:
relocatable definition: GGT_code = 00000000  (0)
relocatable definition: GGT_AA = 00000000  (0)
 8 bit references on: GGT_AD
00000000: 00000019 ....
16 bit references on: GGT_AD
00000000: 00000028 ...(
32 bit references on: OberonLib_vars
00000000: 00000002 0000001C 00000022 ..........."
HUNK_END
```

Der nächste Hunk enthält lediglich einen Zeiger auf den Speicherbereich der globalen Variablen dieses Moduls:

```
HUNK_UNIT: GGT
HUNK_NAME: GGT
HUNK_CODE: 00000001  (1)
00000000: 00000000              ORI.B       #$0000,D0
HUNK_EXT:
relocatable definition: GGT_AB = 00000000  (0)
32 bit references on: GGT_vars
00000000: 00000000 ....
HUNK_END
```

Hier beginnt nun das eigentliche Programm, der Code der für die Prozedur ggt() erzeugt wurde:

```
HUNK_UNIT: GGT
HUNK_NAME: GGT
HUNK_CODE: 00000002 (2)
00000000: BE46              CMP.W     D6,D7
00000002: 6F04              BLE       $00000008
00000004: 9E46              SUB.W     D6,D7
00000006: 60F8              BRA       $00000000
00000008: BC47              CMP.W     D7,D6
0000000A: 6F04              BLE       $00000010
0000000C: 9C47              SUB.W     D7,D6
0000000E: 60F0              BRA       $00000000
00000010: 3007              MOVE.W    D7,D0
00000012: 4E75              RTS
HUNK_EXT:
relocatable definition: GGT_AC = 00000000 (0)
HUNK_END
```

Der letzte Code-Hunk enthält den BEGIN- und den CLOSE-Teil des Moduls GGT. Hier gibt es Referenzen auf die Prozeduren io.WriteString, io.ReadIntegerOk, io.WriteInt, io.WriteLn und auf die BEGIN- und CLOSE-Teile der importierten Module (io und OberonLib):

```
HUNK_UNIT: GGT
HUNK_NAME: GGT
HUNK_CODE: 00000003 (3)
00000000: 2A7A0000          MOVEA.L   $00000002(PC),A5
00000004: 526D0004          ADDQ.W    #1,4(A5)
00000008: 0C6D00010004      CMPI.W    #$0001,4(A5)
0000000E: 665A              BNE       $0000006A
00000010: 4EBA0000          JSR       $00000012(PC)
00000014: 2A7A0000          MOVEA.L   $00000016(PC),A5
00000018: 284D              MOVEA.L   A5,A4
0000001A: 429C              CLR.L     (A4)+
0000001C: 487900000000      PEA       $00000000
00000022: 3F3C0005          MOVE.W    #$0005,-(A7)
00000026: 4EBA0000          JSR       $00000028(PC)
0000002A: 486D0002          PEA       2(A5)
0000002E: 4EBA0000          JSR       $00000030(PC)
00000032: 487900000006      PEA       $00000006
00000038: 3F3C0005          MOVE.W    #$0005,-(A7)
0000003C: 4EBA0000          JSR       $0000003E(PC)
00000040: 4855              PEA       (A5)
00000042: 4EBA0000          JSR       $00000044(PC)
00000046: 48790000000C      PEA       $0000000C
0000004C: 3F3C000C          MOVE.W    #$000C,-(A7)
```

```
00000050: 4EBA0000            JSR       $00000052(PC)
00000054: 3E2D0002            MOVE.W    2(A5),D7
00000058: 3C15                MOVE.W    (A5),D6
0000005A: 61FF                BSR       $0000005B
0000005C: 48C0                EXT.L     D0
0000005E: 2F00                MOVE.L    D0,-(A7)
00000060: 4267                CLR.W     -(A7)
00000062: 4EBA0000            JSR       $00000064(PC)
00000066: 4EBA0000            JSR       $00000068(PC)
0000006A: 4E75                RTS
0000006C: 2A7A0000            MOVEA.L   $0000006E(PC),A5
00000070: 536D0004            SUBQ.W    #1,4(A5)
00000074: 66F4                BNE       $0000006A
00000076: 4EFA0000            JMP       $00000078(PC)
0000007A: 4E71                NOP
HUNK_EXT:
relocatable definition: GGT_V12CA049501C7_open = 00000000  (0)
relocatable definition: GGT_close = 0000006C  (108)
relocatable definition: GGT_AD = 00000000  (0)
16 bit references on: GGT_AB
00000000: 0000006E 00000016 00000002 ...n........
 8 bit references on: GGT_AC
00000000: 0000005B ...[
32 bit references on: GGT_data
00000000: 0000001E 00000034 00000048 .......4...H
16 bit references on: io_V12CC009F0A07_open
00000000: 00000012 ....
16 bit references on: io_close
00000000: 00000078 ...x
16 bit references on: io.WriteLn
00000000: 00000068 ...h
16 bit references on: io.WriteInt
00000000: 00000064 ...d
16 bit references on: io.WriteString
00000000: 00000052 0000003E 00000028 ...R...>...(
16 bit references on: io.ReadInteger
00000000: 00000044 00000030 ...D...0
HUNK_END
```

Der folgende BSS-Hunk besorgt den Speicher für die globalen Variablen dieses Moduls:

```
HUNK_UNIT: GGT
HUNK_NAME: GGT
HUNK_BSS: 00000004  (4)
Size: 00000002  (2)
HUNK_EXT:
```

```
relocatable definition: GGT_vars = 00000000  (0)
HUNK_END
```

Der letzte Daten-Hunk enthält die Zeichenkettenkonstanten, die im Modul GGT definiert wurden:

```
HUNK_UNIT: GGT
HUNK_NAME: GGT
HUNK_DATA: 00000005  (5)
00000000: 61203D20 00006220 3D200000 47475428 a = ..b = ..GGT(
00000010: 612C6229 203D2000 a,b) = .
HUNK_EXT:
relocatable definition: GGT_data = 00000000  (0)
HUNK_END
HUNK_END
```

19. Das Tool XRef

Dieses Programm erzeugt ein Cross-Referenz-Listing der in einem Modul definitierten Bezeichner. Die Bezeichnernamen werden alphabetisch sortiert und zusammen mit der Nummer der Zeilen, in denen sie deklariert wurden, ausgeben. Man hat somit eine kurze Zusammenfassung eines Moduls und kann schnell die Position der Bezeichner im Quelltext finden.

Aufruf vom CLI:

Aufruf:

XRef {<Quelltext>}

Es können beliebig viele Quelltexte übergeben werden. Wird XRef ohne Parameter gestartet, wartet es wie der Compiler auf die Eingabe eines Quelltextnamens.

Aufruf von der Workbench:

Von der Workbench aus kann XRef wie der Compiler durch Anklicken der Quelltexte und Doppelklicken von XRef bei gedrückter Shift-Taste gestartet werden.

Arbeitsweise von XRef:

Nachdem XRef den Quelltext eingelesen hat, werden alle exportierten Bezeichner bestimmt. Die Liste dieser Bezeichner wird alphabetisch sortiert und mit der Endung '.xref' in das Verzeichnis des Quelltextes gespeichert.

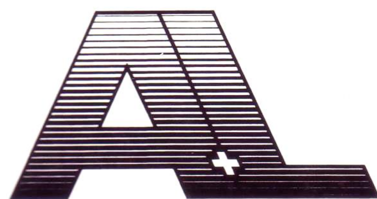
Beispiel:

Das Cross-Reference-Listing des Standardmoduls Lists wird beim Aufruf von 'XRef Lists.mod' erzeugt. Es enthält folgenden Text:

```
Oberon Cross-Reference of om:Lists.mod

Identifier          Line

AddBefore           79
AddBehind           92
AddHead             35
AddTail             45
CountElements       111
DoBackward          132
DoForward           123
DoProc              22
Empty               105
Head                167
Init                27
List                16
Next                141
Node                13
NodePtr             12
Pred                161
Previous            148
RemHead             65
RemTail             72
Remove              55
Succ                155
Tail                173
```



A + L AG
Däderiz 61
CH-2540 Grenchen